NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

THE DESIGN OF A PROGRAMMABLE
CONVOLUTIONAL ENCODER USING
VHDL AND AN FPGA

by

Andrew H. Snelgrove

December 1994

Thesis Co-Advisors: Chin-Hwa Lee
                                   Herschel Loomis

Approved for public release; distribution is unlimited

19950125 165

DISCLAIMER NOTICE

DEFENSE
TECHNICAL
INFORMATION
CENTER
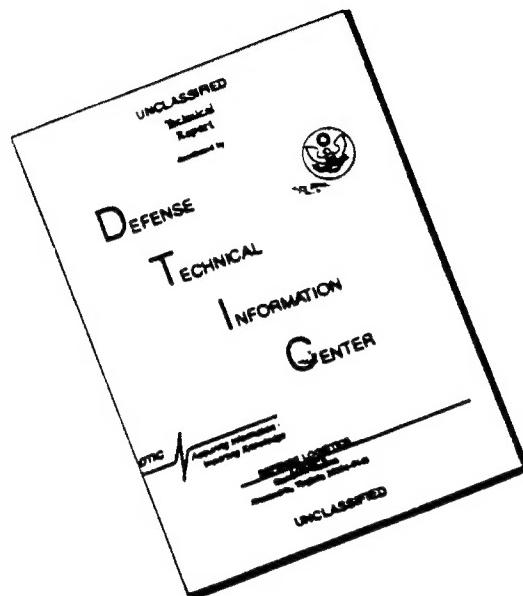
THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REFORT **Approved for public release; distribution is unlimited.**
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (If applicable) ECE	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
THE DESIGN OF A PROGRAMMABLE CONVOLUTIONAL ENCODER USING VHDL AND AN FPGA

12. PERSONAL AUTHOR(S)
Snelgrove, Andrew H.

13a. TYPE OF REPORT Masters Thesis	13b. TIME COVERED FROM TO	14. DATE OF REPORT (Year,Month,Day) December 1994	15. PAGE COUNT 120

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Convolutional encoding, VHDL, FPGA, top-down design, one-hot state assignment

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Convolutional encoding is a Forward Error Correction (FEC) technique used in continuous one-way and real time communication links. It can provide substantial improvement in bit error rates so that small, low power, inexpensive transmitters can be used in such applications as satellites and hand-held communication devices. This thesis documents the development of a programmable convolutional encoder implemented in a Field Programmable Gate Array (FPGA) from Xilinx, Inc., called the XC3064 Logic Cell Array (LCA). The encoder is capable of coding a digital data stream with any one of 39 convolutional codes. Because the LCA is used for the hardware implementation, the design can be changed or expanded conveniently in the lab. In particularly flexible systems, several encoder designs can be stored in the system RAM, each one being downloaded into the LCA under different circumstances. The encoder has a simple microprocessor interface, a register file for storage of code parameters, a test circuit, and a maximum bit rate of about 15 Mbits/s. Special design techniques like one-hot state assignment, pipelining, and the use of redundant states are employed to tailor the hardware to the LCA architecture. Other ways to improve the output bit rate are suggested. The VHSIC Hardware Description Language (VHDL) is used to model abstract behavior and to define relationships between building blocks before the hardware implementation phase.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Lee, Chin-Hwa	22b. TELEPHONE (Include Area Code) 408-656-2190 — 22c. OFFICE SYMBOL EC/Le

THE DESIGN OF A PROGRAMMABLE CONVOLUTIONAL ENCODER USING VHDL AND AN FPGA

by

Andrew H. Snelgrove
Naval Air Warfare Center - Weapons Division
B.S., Rensselaer Polytechnic Institute, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
from the
NAVAL POSTGRADUATE SCHOOL

December 1994·

Author: _____
Andrew H. Snelgrove

Approved by: _____
Chin-Hwa Lee, Thesis CoAdvisor

_____
Herschel H. Loomis, Thesis CoAdvisor

_____
Michael A. Morgan, Chairman,
Department of Electrical and Computer Engineering

ii

ABSTRACT

Convolutional encoding is a Forward Error Correction (FEC) technique used in continuous one-way and real time communication links. It can provide substantial improvement in bit error rates so that small, low power, inexpensive transmitters can be used in such applications as satellites and hand-held communication devices. This thesis documents the development of a programmable convolutional encoder implemented in a Field Programmable Gate Array (FPGA) from Xilinx, Inc., called the XC3064 Logic Cell Array (LCA). The encoder is capable of coding a digital data stream with any one of 39 convolutional codes. Because the LCA is used for the hardware implementation, the design can be changed or expanded conveniently in the lab. In particularly flexible systems, several encoder designs can be stored in the system RAM, each one being downloaded into the LCA under different circumstances. The encoder has a simple microprocessor interface, a register file for storage of code parameters, a test circuit, and a maximum bit rate of about 15 Mbits/s. Special design techniques like one-hot state assignment, pipelining, and the use of redundant states are employed to tailor the hardware to the LCA architecture. Other ways to improve the output bit rate are suggested. The VHSIC Hardware Description Language (VHDL) is used to model abstract behavior and to define relationships between building blocks before the hardware implementation phase.

Accesion For		
NTIS CRA&I		☒
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I. INTRODUCTION

Convolutional encoding is a method of adding redundancy to a data stream in a controlled manner to give the destination the ability to correct bit errors without asking the source to retransmit. Convolutional codes, and other codes which can correct bit errors at the receiver, are called *forward error correcting* (FEC) codes. Contrast convolutional encoding with the common *automatic repeat request* (ARQ) error correction schemes which require a second communication channel between the source and destination. The receiver requests retransmissions from the source when it detects a bit error. The added delays due to retransmission requests and the actual retransmissions degrade the throughput of the communication link. Convolutional codes add reliability to the link while eliminating the need for a reverse channel. They are used in applications where retransmission of data is impractical or impossible, such as in space probes, or in broadcast satellites that transmit to multiple receivers simultaneously (Stallings, 1994, p. 149), or in real time speech transmissions.

This thesis leads the reader through the entire design cycle of a programmable convolutional encoder that can be utilized in many different systems that use various convolutional codes. First, it explains the basics and advantages of convolutional encoding in Chapter II. Chapter III then describes the top-down design paradigm and breaks down the programmable encoder design into smaller building blocks, detailing the behavior of each block as it proceeds. After all of the blocks and their interconnectivity and interaction are defined, the chapter concludes with an example of the blocks working together as one unit. Chapter IV covers the VHDL source code used to model and simulate the encoder in an abstract, behavioral context. No hardware details are defined. Chapter V discusses Field Programmable Gate Arrays (FPGAs) in general, their limitations, and their advantages. It

then gives a detailed treatment of the Xilinx XC3000 family of "Logic Cell Array", Xilinx Corp.'s name for "FPGA". The design described in this thesis will be implemented in a Xilinx XC3064 LCA. After the peculiarities of FPGAs are described, Chapter VI explains the one-hot state assignment technique, why it works better than conventional highly encoded state assignments, and how to use it to force flip-flop fan-in logic into a single FPGA logic block. The chapter uses one of the state machines of the encoder design to illustrate the method. Finally, Chapter VII describes the Xilinx development system and its various CAD programs. It also explains how simulation was used to estimate speed performance of the design and how pipeline registers were inserted into several blocks to improve combinational delay times and hence clock rate. The chapter concludes with a few comments about the Mentor Graphics Autologic tool and about the idea of back annotating performance data into the VHDL code. Neither the Autologic tool nor the back annotating idea proved useful in this work.

II. CONVOLUTIONAL ENCODING

This chapter presents the basics of convolutional encoding including its location in the communication link, how convolutional codes are described and implemented, and the benefits of using them.

A. INTRODUCTION

Figure 2.1 shows a basic communication link using convolutional encoding. An information source generates a sequence of message bits, **m**, and feeds them into a convolutional encoder. The encoder produces a sequence of coded bits, **U**, which modulates a waveform. The waveform, $s_i(t)$, travels through a channel where it is corrupted by additive white Gaussian noise (AWGN). The corrupted signal, $s'_i(t)$, is demodulated to recover the coded bits, **Z**, which contain bit errors because of the AWGN. The convolutional decoder then takes advantage of the redundancy added by the code to correct bit errors, producing an estimate of the original bit stream, **m'**. The estimate is very close, if not identical, to the original, **m**. (Sklar, 1988)
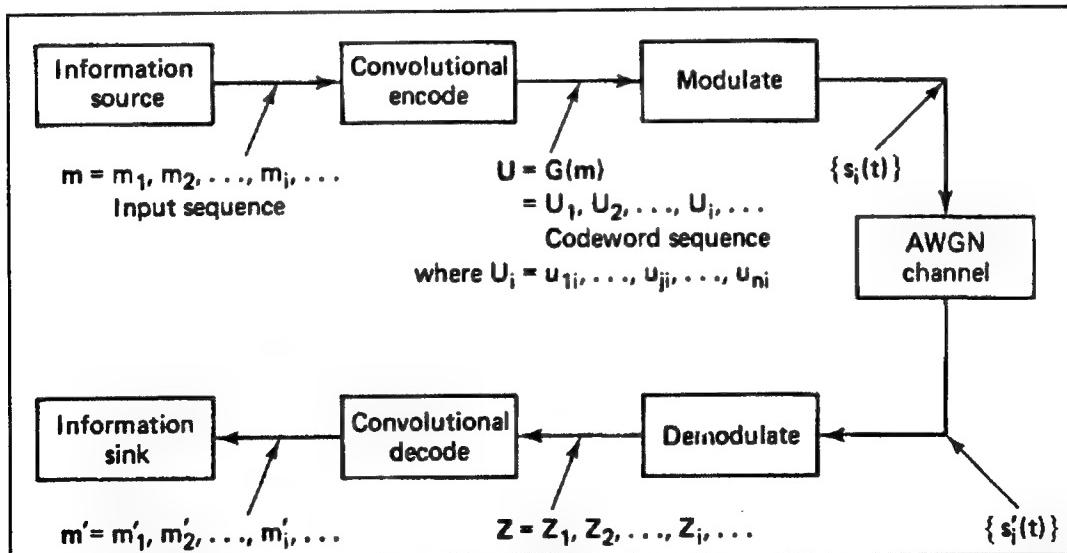
Figure 2.1. Relationship of encoding/decoding in a communication link.
(Sklar, 1988, p. 316)

3

B. CONVOLUTIONAL CODES

Convolutional codes are forward error correcting codes which take a group of k information or message bits, called a *k-tuple,* and maps it into another group of n code bits called an *n-tuple.* Each n-tuple is determined by the most recently arrived k-tuple and the *L-1* previously arrived k-tuples. The codes are described by a fraction k/n, called the *rate,* and L, called the *constraint length.*

C. ENCODERS

Figure 2.2 shows the structure of a convolutional encoder. Message bits arrive as k-tuples at one end of a series of L serial shift register stages, each stage holding one k-tuple. Each of the L k-tuples helps determine the n-tuple. With more stages (i.e., a longer constraint length, L), each k-tuple influences more n-tuples, increasing the amount of redundancy contained in the output coded bit stream. The parallel outputs of the registers feed n modulo-2 adders via a bank of AND gates, the purpose of which is described in the
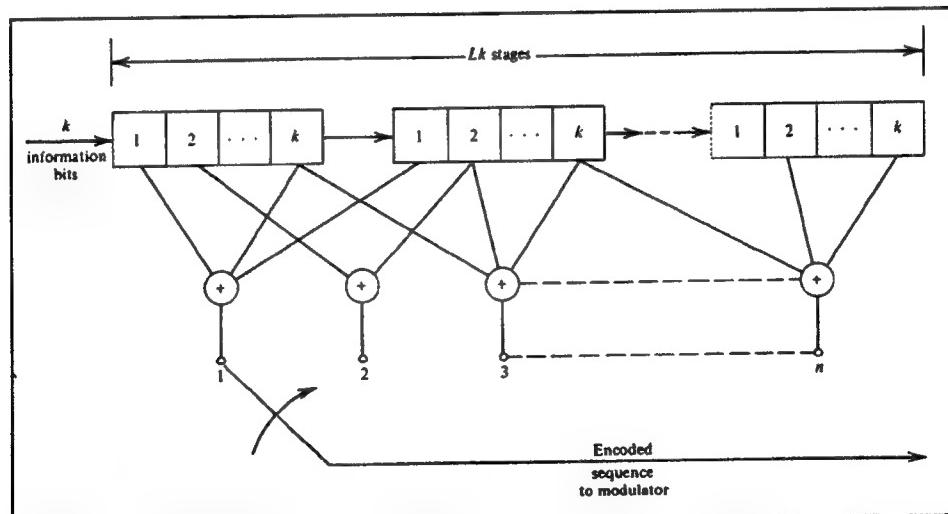
Figure 2.2. Generic convolutional encoder. (Proakis, 1989, p. 443)

next section. Each adder consists of an Lk-input XOR tree symbolized by the circled "plus"

sign and outputs one bit of the n-tuple. The one-bit outputs of the n modulo-2 adders are delivered sequentially to the modulator as a convolutionally encoded bit stream.

Figure 2.3 shows an example of a rate 2/3 convolutional encoder with a constraint length $L = 2$. Notice the following: (1) the encoder has two stages because $L = 2$, (2) each stage holds two bits because $k = 2$, and (3) there are three modulo-2 adders because $n = 3$.

D. CONNECTION VECTORS

In Figures 2.2 and 2.3, a solid line or arrow between a shift register and a modulo-2 adder represents a connection between the corresponding bits. Thus, in Figure 2.3, for adder number 3 there is a connection between shift register bit number 3 and adder input bit number 3. Similarly, there is also a connection between shift register bit number 1 and adder input bit number 1. The serial input end of the shift register is considered most significant.
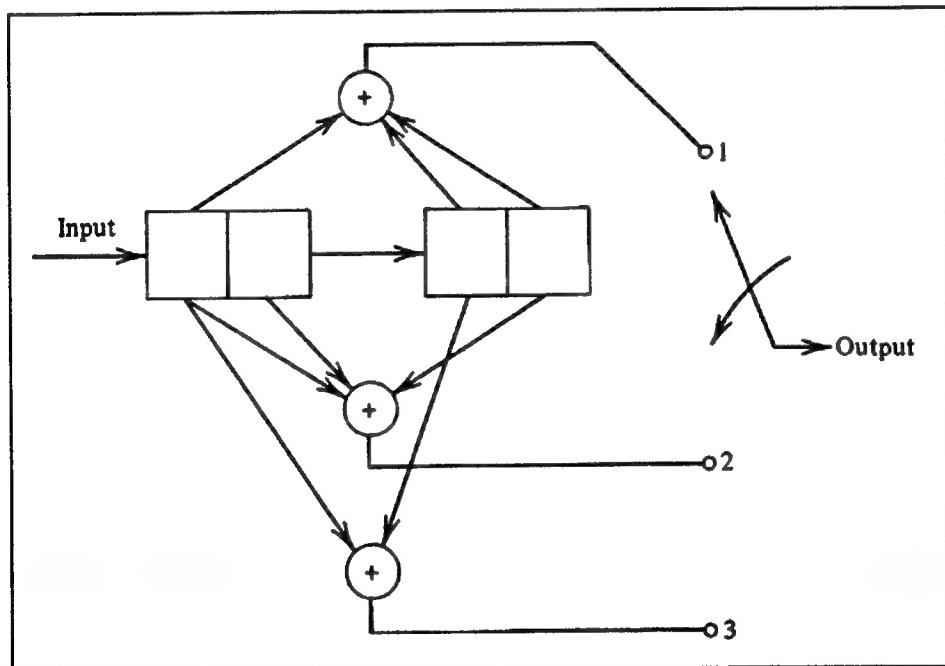
Figure 2.3. Rate 2/3, $L = 2$, convolutional encoder. (Proakis, 1989, p. 445)

An absence of a line or arrow indicates that there is no connection. In practice, the connectivity between the shift register and the adders is expressed with *connection vectors*. Each adder has an Lk-bit connection vector, **g**, associated with it (Sklar, 1988, p. 318). A '1' in the i-th position of the vector represents a connection between the i-th bit of the shift register and the i-th bit of the adder input, whereas a '0' represents no connection. Connection vectors are written as octal numbers. In Figure 2.3, there are three 4-bit connection vectors: (1) $g_1 = 1011$, written as 13_8, (2) $g_2 = 1101$, written as 15_8, and (3) $g_3 = 1010$, written as 12_8. In hardware, the contents of the shift register is bitwise ANDed with all of the n connection vectors. The outputs of the AND gates are then modulo-2 added together to arrive at a single bit of the n-tuple. Thus, each adder has Lk AND gates feeding its inputs. Figure 2.4 shows the hardware needed to generate one bit of the n-tuple.

E. CODING GAIN

Coding gain is the difference in E_b/N_0 required to achieve the same probability of bit error, P_B, between a coded transmission and an uncoded transmission over the same channel using the same modulation technique (Sklar, 1988, p. 345). E_b is the average signal energy per bit (Sklar, 1988, p. 156) and N_0 is noise power spectral density of white noise (Sklar, 1988, p. 345). Table 2.1 lists the required E_b/N_0 to achieve three different values of P_B. For each P_B, it also lists the coding gain provided by various convolutional code rates and constraint lengths. Constraint length is denoted by K rather than L in this table.

Significant coding gain can be achieved with a fairly simple code. For example, Table 2.1 lists a gain of 6.2 dB when a rate 1/3, constraint length 7, code is used to achieve a probability of bit error of 10^{-7}. Without coding, the required E_b/N_0 to achieve the same P_B is 11.3 dB; whereas, with coding, the required E_b/N_0 drops by 6.2 dB. This implies that the required transmitter power for the coded communication link is less than 25% of the transmitter power needed for the uncoded link to achieve the same probability of bit error.
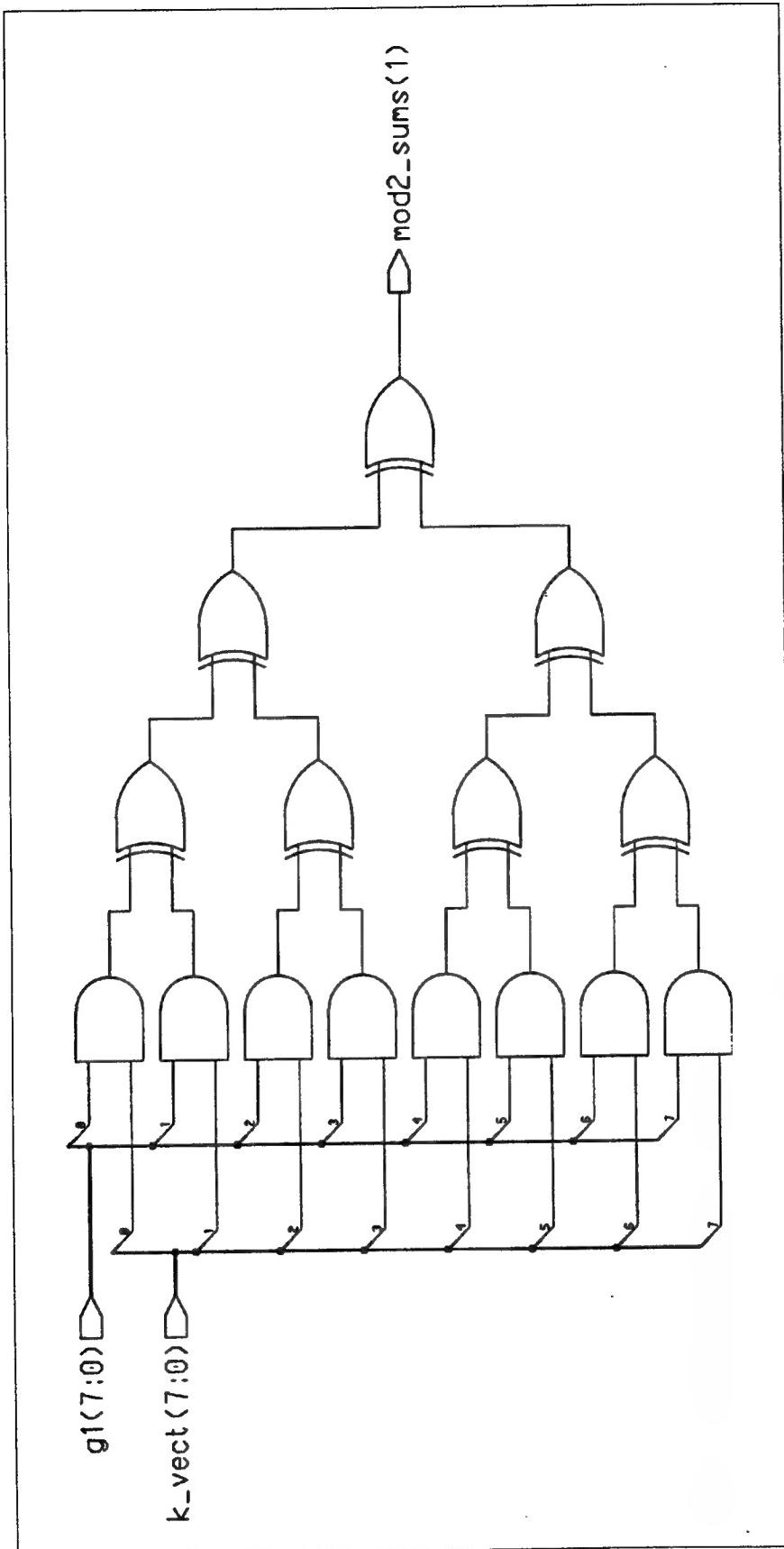
Figure 2.4. Hardware necessary to generate one code bit.

TABLE 2.1. CODING GAINS (dB) FOR SEVERAL BIT ERROR PROBABILITIES.
(Sklar, 1988, p. 346)

Uncoded E_b/N_o (dB)	Code rate		↓		↓			↓		↓	
	P_B	K	7	8	5	6	7	6	8	6	9
6.8	10^{-3}		4.2	4.4	3.3	3.5	3.8	2.9	3.1	2.6	2.6
9.6	10^{-5}		5.7	5.9	4.3	4.6	5.1	4.2	4.6	3.6	4.2
11.3	10^{-7}		6.2	6.5	4.9	5.3	5.8	4.7	5.2	3.9	4.8
Upper bound			7.0	7.3	5.4	6.0	7.0	5.2	6.7	4.8	5.7

In a more practical sense, lower transmitter power implies smaller, lighter, cooler, more reliable, portable electronics packages.

Notice the trends in the gains listed in the table. Coding gain increases for the lower code rates because the proportion of output coded bits to input message bits is larger. A higher proportion places more redundancy in the coded bit stream. Consequently, a lower transmitter power is adequate for the same probability of bit error, and, therefore, a higher coding gain is established. Similarly, within a code rate, coding gain increases with constraint length because longer constraint lengths imply that more k-tuples affect each n-tuple. With each n-tuple determined by a larger set of k-tuples, more redundancy is added to the coded bit stream.

III. ENCODER DESIGN DETAILS

This chapter begins the discussion of the programmable convolutional encoder design. First, it touches on the top-down design paradigm and then presents a list of useful features which a programmable encoder should have. Second, it shows how the design is partitioned into building blocks providing a detailed behavioral description of each block. Finally, the chapter uses an example code rate of 3/5 and a simplified timing diagram to describe how the blocks interact to produce a complete encoder.

A. TOP-DOWN DESIGN

The top-down approach allows the designer to simulate, debug, and evaluate different variations of the overall design without implementing anything in hardware. It begins with describing a system's behavior at a high level of abstraction without regard to any hardware details. As the system develops, each subsystem is broken down into a hierarchy of ever smaller and simpler building blocks, all of whose behavior is defined abstractly with a hardware description language (HDL). Hardware considerations such as target technology, state assignments, etc., are not important at this stage. Only the functional descriptions of the blocks and their interconnectivity matter. When all blocks and their interactions have been defined, hardware implementation of each block proceeds.

The benefit of the approach is that designs are evaluated without bogging down in hardware details, and system level bugs can be discovered and fixed early in the design cycle. In addition, the HDL code and simulator output waveforms document the required behavior of the target hardware. They also provide a means of documenting upgrades to the system throughout the system's lifecycle.

B. TOP-LEVEL OVERVIEW

There are three main constraints imposed on the programmable convolutional encoder design: (1) it must have the ability to encode serial data streams with many combinations of k, n, and L, (2) its coding parameters must be adjustable via a microprocessor data bus, and (3) it must be implemented in a Field Programmable Gate Array. Absolute data rate is not a concern for the purposes of this thesis, but the implementation takes advantage of the FPGA architecture to enhance speed. FPGA implementation is covered in a later chapter.

Given the first two design requirements and the structure of a generic convolutional encoder, the design must have, at a minimum, the following architectural features:

1. An 8-bit data bus and a handshaking mechanism for writing code parameters to the device,
2. a register file to hold code parameters,
3. one AND/XOR tree for each connection vector,
4. an input shift register for message bits,
5. a special shift register to shift k-tuples of message bits through the encoder,
6. a state machine to control the incoming message bits,
7. a state machine to control the k-tuple shift register,
8. a state machine to control the outgoing coded bits, and
9. a test circuit and input multiplexor.

Figure 3.1 shows a block diagram of the design. It consists of nine sub-blocks: IN_ENBLE, SHIFTREG, LOADER, DATAREG, GENERATOR, SEQUENCER, TEST, MUX, and REGFILE. The architectural features listed above reside in the block that has a corresponding number in the lower left corner.

The datapath blocks are clocked on the falling edge of the system clock, while the control blocks (dotted outlines) are clocked on the rising edge. Both clock edges are used to make the output bit rate equal to the clock frequency. If one edge was used exclusively, then the clock frequency would have to be doubled to get the same output bit rate. The

10

encoder has a global asynchronous reset, "reset", and a global clock, "clk", not shown in Figure 3.1.
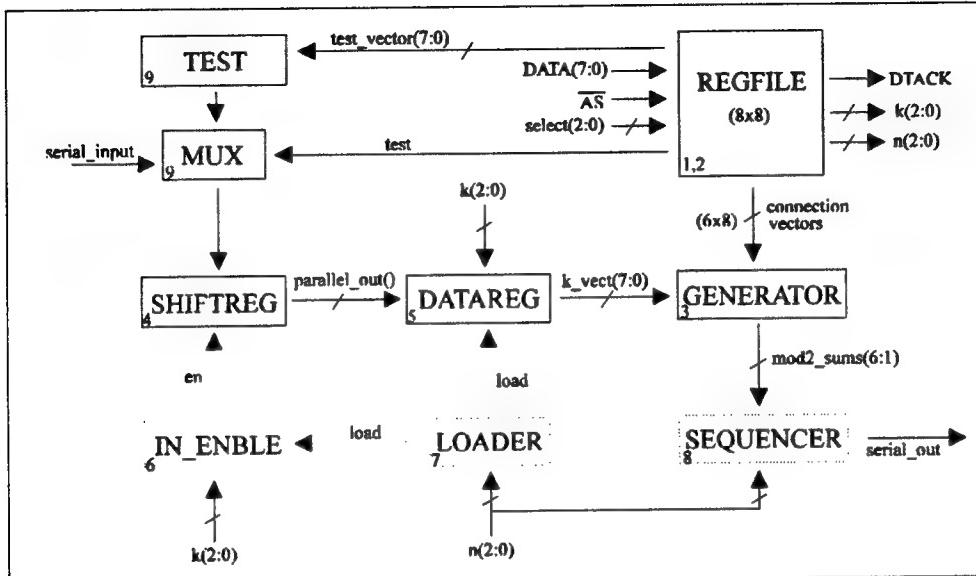
Figure 3.1. Block diagram of the convolutional encoder.

Because of their serial nature, convolutional encoders lend themselves well to a pipeline architecture that allows the encoder to input message bits, convert them to code bits, and output them simultaneously. The idea behind pipelining is to have several independent stages working on different sets of data concurrently. The output of one stage becomes the input to the next stage. In this convolutional encoder design, serial message bits move through the SHIFTREG block and land in the DATAREG block while the SEQUENCER block sends code bits from the GENERATOR block to the "serial_out" port. Detailed descriptions of these blocks appear later in this chapter.

C. DATAPATH

The datapath consists of the blocks SHIFTREG, DATAREG, and GENERATOR. Refer to Figure 3.1. Although SEQUENCER is considered a control block, it does serve a datapath function because it acts like a multiplexor that selects bits from "mod2_sums(6:1)".

11

The details of SEQUENCER are presented in a later section.

In a nutshell, the datapath operates as follows. SHIFTREG, a serial to parallel shift register latches k serial message bits. At the appropriate time, these message bits are placed into DATAREG in parallel where they become part of "k_vect(7:0)", the DATAREG output. GENERATOR combines "k_vect(7:0)" with six connection vectors and delivers six bits in parallel, "mod2_sums(6:1)", to SEQUENCER. Based on the value of "n(2:0)", SEQUENCER selects the appropriate bits and sends them serially to the output, "serial_out". What follows detailed behavioral description of each datapath block.

1. MUX

The MUX block is a 2-to-1 multiplexor which selects either "serial_input" or a test pattern to be the serial input of SHIFTREG. Its selection control signal is "test" which is one bit of one of the registers in REGFILE. When "test" is high, the test pattern is fed into SHIFTREG; otherwise, "serial_input" is fed into SHIFTREG.

2. SHIFTREG

This block is an ordinary 4-bit serial-in/parallel-out shift register. Its inputs are "serial_input", "reset", "en", and "clk". Message bits enter the convolutional encoder through "serial_input". "En" enables SHIFTREG long enough to shift a k-tuple of message bits. SHIFTREG's outputs, "parallel_out[4..1]", are simply the parallel version of the serial input, and they feed the four parallel inputs of DATAREG.

3. DATAREG

The DATAREG block is a specialized 8-bit shift register which loads and shifts its input in k-tuples. In effect, the contents shift by one k-tuple with a single clock edge. For example, if $k = 2$, then bits 4 and 5 shift two places to become bits 6 and 7, 2 and 3 become 4 and 5, 0 and 1 become 2 and 3, and the next two input bits become bits 0 and 1. DATAREG's inputs are "load", "k(2:0)", "clk", "reset", and "in(7:4)", and its output is

12

"k_vect(7:0)". "K(2:0)" is a binary number representing the number of bits per *k*-tuple. "In(7:4)" are the outputs from SHIFTREG. In the block diagram, "in(7:4)" is shown as "parallel_out()" because two different port names were used for the same signals as VHDL source code was developed. Ideally, only one name should be used. "K_vect(7:0)" is the 8-bit contents of DATAREG, which, along with the connection vectors of the GENERATOR block, determines the coded bits in "mod2_sums(6:1)".

4. GENERATOR

GENERATOR is the only combinational logic block of the encoder design. It calculates the "mod2_sums(6:1)" vector which determines the output code sequence. GENERATOR's inputs are the six connection vectors, "g1" through "g6" and "k_vect(7:0)" from DATAREG. It logical ANDs each of the six connection vectors with "k_vect(7:0)" and modulo-2 adds (XORs) the elements of each resulting vector to produce the "mod2_sums(6:1)" vector for the SEQUENCER block. Note that the constraint length, L, is inherent in the choice of generator vectors. Because they are eight bits wide, they can provide a constraint length of 8 or less for $k = 1$, 4 or less for $k = 2$, and 1 or 2 for either $k = 3$ or $k = 4$.

5. REGFILE

This block is a register file with eight 8-bit registers and a state machine, HANDSHAK, that provides the handshaking mechanism. Six of the registers, register 1 through register 6, hold the six connection vectors. Register 0 holds "k(2:0)" in bits 2 through 0, "n(2:0)" in bits 5 through 3, and the "test" control bit in bit 6. Each register enable comes from a 3-to-8 decoder output which is the decoded equivalent of the 3-bit "select(2:0)" bus. The address strobe, AS, uses the enable input on the decoder to allow one of the eight decoder outputs to select the target register.

The handshaking mechanism is controlled by HANDSHAK, a Moore machine

13

clocked on the rising edge of the system clock. The state diagram is shown in Figure 3.2. When AS is asserted (low) by the microprocessor, the state machine output ASout goes high for one clock cycle, enabling the target register for writing. The register
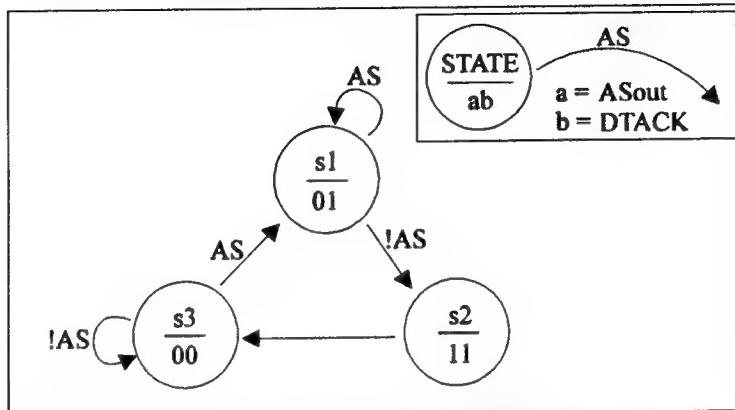
Figure 3.2. State diagram for HANDSHAK.

latches the data on the falling clock edge after which the DTACK signal asserts low and stays there until AS is inactive (high). The handshaking mechanism is patterned after the 68000 family of microprocessors (Clements, 1992, p.181).

D. CONTROL

The control blocks consist of IN_ENBLE, LOADER, and SEQUENCER. LOADER produces the active low signal "load". It divides the clock by n and provides a low pulse every n clock periods. When "load" is active, it allows DATAREG to take in another k-tuple from SHIFTREG. It also gives the IN_ENBLE block a synchronization signal. IN_ENBLE's sole function is to enable SHIFTREG with "en", which stays high long enough for SHIFTREG to input one k-tuple. "En" is active high. SEQUENCER selects the lowest significant n bits from "mod2_sums(6:1)" and sends them to "serial_out", the output of the convolutional encoder. Meanwhile, IN_ENBLE allows the entry into SHIFTREG of the k-tuple that will produce the next n-tuple. When "load" activates again, the new k-tuple is

14

loaded into DATAREG and new values of the "mod2_sums(6:1)" vector appear at SEQUENCER's input.

1. LOADER

LOADER is a Moore machine which divides the system clock by n and provides the "load" signal to DATAREG and IN_ENBLE. Its inputs are "n(2:0)", "clk", and "reset". "N(2:0)" is the binary representation of the number of code bits sent in one n-tuple. It can have a value from two through six. The unused values $n = 0$ and $n = 1$ default to $n = 2$, and the unused value $n = 7$ defaults to $n = 6$. "Load" enables the parallel loading function of DATAREG every n clock cycles. This block must divide the system clock by n because the input bit rate must be multiplied by n to account for the extra bits added to the bit stream in the coding process. Since the output bit rate is fixed at the system clock rate, dividing the
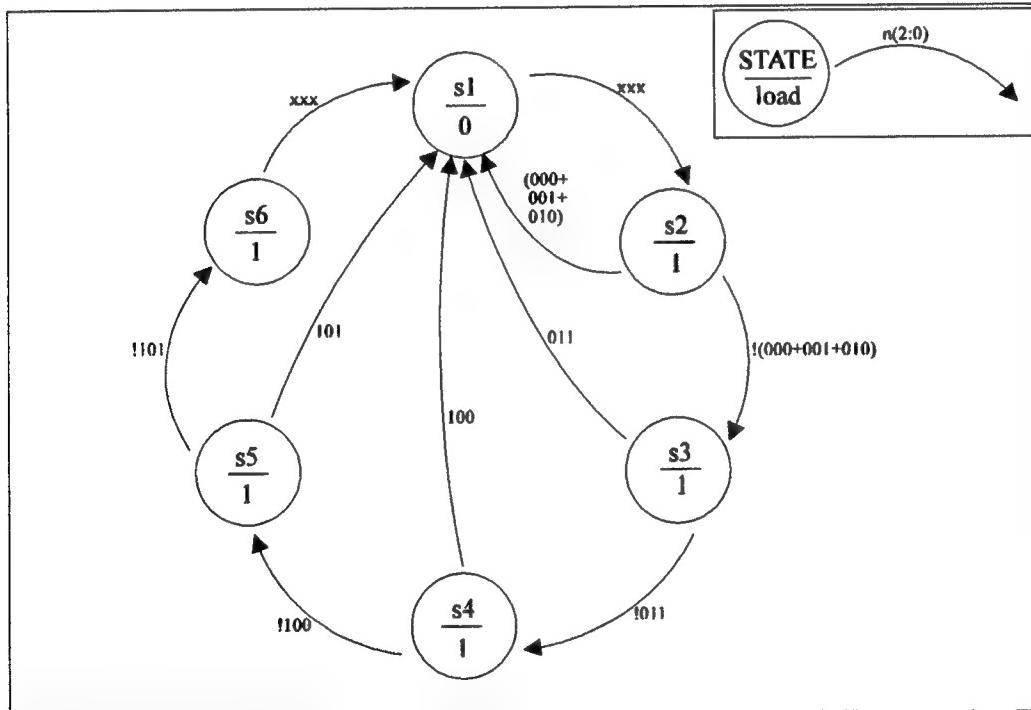
Figure 3.3. State diagram for LOADER block.

clock by n and loading the input bit groups at the divided rate achieves the same result. This

15

approach also eliminates the need for a phase-locked loop clock multiplier for the output bit stream, allowing the design to reside in an FPGA.

Figure 3.3 shows the state diagram. The machine resets to s1. The output, "load", is high (inactive) in all states except s1. After n clock transitions, it ends up in s1 with "load" active. For instance, assume the machine is in s1 and $n = 4$. After every fourth rising clock edge, LOADER will be in s1 forcing "load" low. Effectively, LOADER divides the clock rate by four.

2. IN_ENBLE

This block is a Moore machine which counts the number of clock cycles necessary to keep "en" high long enough for SHIFTREG to input k message bits (one k-tuple). IN_ENBLE takes the signals "clk", "load", "reset", and "k(2:0)" as inputs. K can have only four values and could have been encoded in two bits. However, using the 3-bit binary representation for the values of k is less confusing for the user of the encoder and has little cost impact on the overall design. This approach also leaves a bit in place to accommodate future enhancements to the encoder design which could handle more than four message bits. The unused value of $k = 0$ defaults to $k = 1$, and the unused values $k > 4$ default to $k = 4$. "Load" is the output of LOADER. IN_ENBLE uses this signal to synchronize its activity with LOADER.

As the state diagram, Figure 3.4, shows, IN_ENBLE stays in state s0 with "en" inactive until the "load" signal is active (low). This feature guarantees that IN_ENBLE allows SHIFTREG to begin taking new input immediately after "load" latches a k-tuple into DATAREG. Once "load" is active, IN_ENBLE activates "en" for k clock cycles. Within these k cycles are k falling clock edges that SHIFTREG uses to latch k message bits.
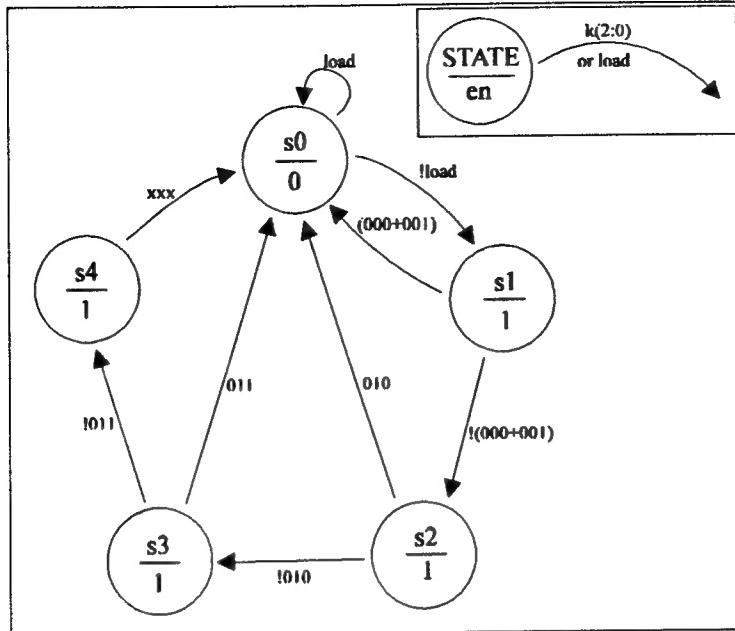
Figure 3.4. State diagram for IN_ENBLE block.

3. SEQUENCER

The SEQUENCER block is a Mealy machine that traverses through n states selecting bits from "mod2_sums(6:1)" for output from the convolutional encoder. "mod2_sums(6:1)" is the vector containing the six modulo-2 sums resulting from the GENERATOR block. The block's other inputs are "n(2:0)", "reset", "load", and "clk". "n(2:0)" is a binary number representing the number of code bits in one n-tuple.

Figure 3.5 shows the SEQUENCER state diagram. It selects the output bits in order from low index to high index. Thus, if $n = 3$, the state machine will traverse through states s1, s2, and s3 repeatedly, selecting the correspondingly indexed bit from the "mod2_sums(6:1)" vector. In this case, it would select "mod2_sums(1)", "mod2_sums(2)", and "mod2_sums(3)". As in the LOADER block, the unused values $n = 0$ and $n = 1$ default to $n = 2$, and the unused value $n = 7$ defaults to $n = 6$.

17
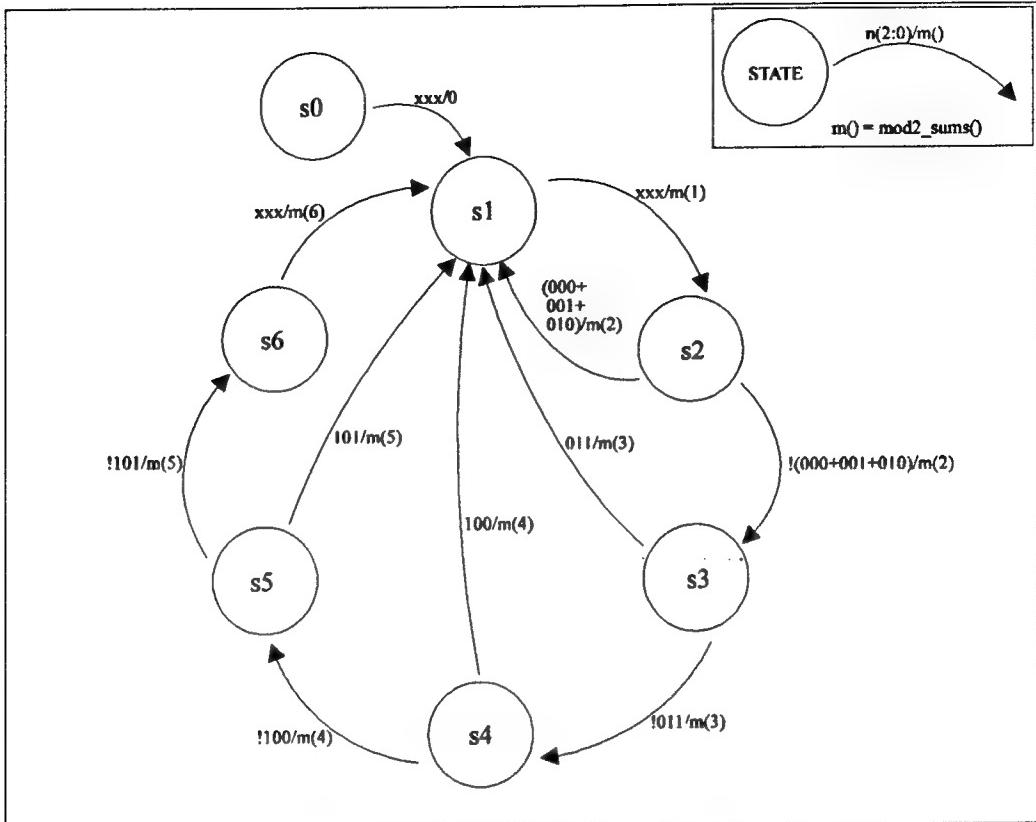
Figure 3.5. State diagram for SEQUENCER block.

4. TEST

The TEST block consists of an 8-to-1 multiplexor whose selection and enable bits are controlled by a 4-bit binary counter. The counter cycles through the eight inputs of the multiplexor on its first eight of sixteen state transitions. It forces the multiplexor output to zero during the second eight transitions, filling SHIFTREG with zeros. This is needed to obtain the correct output sequence that corresponds to the input test pattern. The eight inputs to the multiplexor come from "test_vector(7:0)" which is the test pattern stored in register 7 of REGFILE.

E. OPERATION

Figure 3.6 shows a simplified timing diagram to clarify the operation of the encoder. The vertical lines lettered A through J correspond to each step the encoder executes as it inputs

18

a k-tuple and simultaneously outputs an n-tuple. For this example, the encoder is set up for a 3/5 code, and the k-tuple entering the encoder is "101". The MSB enters first. At the start of the example, the contents of DATAREG and SHIFTREG are all 'x's representing k-tuples which have not yet completed their journey through the encoder. This example looks at the operation of the encoder in midstream.
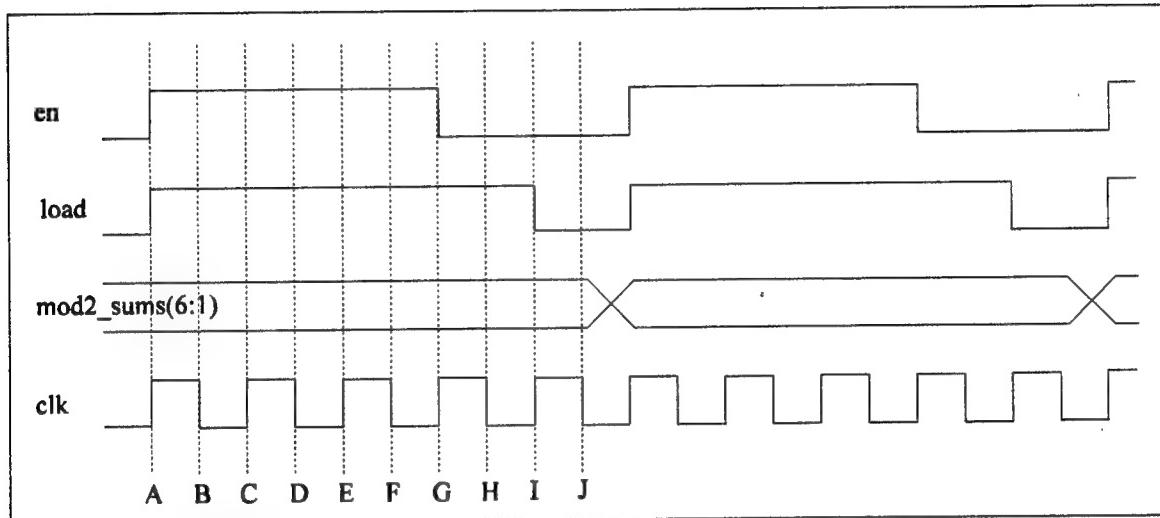
Figure 3.6. Simplified timing diagram for the convolutional encoder set up for a 3/5 code.

A. DATAREG is disabled ("load" is high), SHIFTREG is enabled ("en" is high), and all state machines advance 1 state. The contents of DATAREG and SHIFTREG are all 'x's (previous bit arrivals).

B. SHIFTREG latches the 1st bit of the k-tuple ('1'), SEQUENCER outputs the 1st bit of the previous n-tuple (mod2_sums(1)). The contents of SHIFTREG is now "1xxx".

C. All state machines advance 1 state.

D. SHIFTREG latches the 2nd bit of the k-tuple ('0'), SEQUENCER outputs the 2nd bit of the previous n-tuple (mod2_sums(2)). The contents of SHIFTREG is now "01xx".

E. All state machines advance 1 state.

F. SHIFTREG latches the 3rd and final bit of the k-tuple ('1'), SEQUENCER outputs the 3rd bit of the previous n-tuple (mod2_sums(3)). The contents of SHIFTREG is now "101x".

G. SHIFTREG is disabled ("en" is low), all state machines advance 1 state.

19

H. SEQUENCER outputs the 4th bit of the previous n-tuple (mod2_sums(4)). The contents of SHIFTREG does not changed.

I. DATAREG is enabled ("load" is low), all state machines except IN_ENBLE advance 1 state; IN_ENBLE remains in state s0 waiting for "load" to go high for synchronization of its actions with those of LOADER.

J. DATAREG loads the k-tuple ("101") from SHIFTREG. It now contains "101xxxxx". The new value in DATAREG causes "mod2_sums(6:1)" to change. SEQUENCER outputs the 5th and final code bit from the previous value of mod2_sums(5), completing the output of the previous n-tuple. The encoder goes back to step **A** where it begins outputting the new n-tuple consisting of the lowest significant five bits of the new value of "mod2_sums(6:1)". It also begins inputting a new k-tuple.

F. INTERFACE

The device will need the following pin functions to interface with any external system in which it is a component:

1. system clock, "clk", (input pin),
2. global reset, "reset", (input pin),
3. an input port for serial message bits, "serial_input", (input pin),
4. an output port for serial code bits, "serial_out", (output pin),
5. 8-bit port for the data bus, "data(7:0)", (input pins),
6. address strobe, "AS", used in handshaking (input pin),
7. data transfer acknowledge, "DTACK", for handshaking, (output pin),
8. "en" and "load" for coordinating message input/code output, (output pins)
9. miscellaneous ports for testing/monitoring intermediate signals (output pins).

The user must feed message bits to the encoder in serial bursts so that k bits are available when "en" is high. Therefore, the user should use "en" as a synchronizing input to whatever circuitry precedes the convolutional encoder. An asynchronous I/O First-In-First-Out (FIFO) memory would be the most appropriate structure to hold incoming message bits because it would accept incoming message bits at a constant rate while the encoder removes them in bursts of one k-tuple while "en" is active. For a one-chip solution, a register based FIFO could be implemented in the FPGA along with the encoder design. Thus the input bit

20

stream would be decoupled from the encoder input, and the buffering of message bits would not be the user's worry.

In addition to "en", the signal "load" is also provided as an output for the user to utilize as necessary for interfacing. For troubleshooting purposes, "k_vect(7:0)" and "mod2_sums(6:1)" should also be brought to output pins. These pins can be eliminated after the system is fully tested.

The next chapter deals with VHDL and high level simulation to confirm the correct operation of the programmable convolutional encoder before hardware implementation begins.

IV. VHDL AND SIMULATION

This chapter briefly discusses the VHSIC Hardware Description Language (VHDL) and how it was used to simulate and verify the proper behavior of the encoder design. A few aspects of VHDL dealing with hardware implications of the code and a stimulus block are described. A method to determine correct coded bit sequences is also discussed.

A. VHDL

VHDL is the IEEE and DOD standard for defining system behavior. It has several advantages in support of top-down design. Since it is a standard HDL, it provides a reliable communication medium for transferring design information and specifications between and within design groups. Also, different groups do not need to use the same CAD suite as long as their CAD environment supports VHDL simulation. It is not limited to descriptions of one particular technology, and its wide range of descriptive capability allows one to write accurate models both at the subsystem level and at the gate level. Thus, models of subsystems written on different levels of abstraction can coexist in the same simulation of a system under development. As more detailed models are completed, they can be verified simply by plugging them into the overall system model and resimulating (Lipsett,1989, p.3). No module of the system must be completed before another module can be inserted and debugged because the abstract behavioral models substitute for modules whose hardware details are not yet available. The schematic diagram and VHDL files for the encoder are in Appendix A.

Presenting VHDL in any detail is beyond the scope of this thesis. Therefore, it is assumed that the reader has a basic understanding of the various description styles and syntactical constructs of the language. Of these, only a few are used in the convolutional encoder model, and they are presented along with their hardware embodiments. Keep in

mind that even though a translation is made from VHDL code to a conceptual hardware block, the actual hardware details are still immaterial at this stage in the design cycle. Behavioral description is the only concern. For the interested reader, two good VHDL texts are listed in the List of References. One is Lee, 1992, and the other is Lipsett, 1989.

1. Constructs

Of the three styles of architectures used in VHDL descriptions, the behavioral and dataflow constructs were the only two used. The structural style is most conveniently used at the top level of the design hierarchy. Since this style is basically the text form of a schematic diagram, an actual schematic diagram was chosen as the top level documentation of the encoder design. This approach provides the designer with a convenient graphical format of the design and nicely compliments the trace window of the Mentor Graphics QuicksimII simulator. With the schematic in view inside the QuicksimII environment, graphical blocks can be opened to gain access to internal signals for simulation while keeping the top level schematic in view. Schematic diagrams also make it easier for people unfamiliar with the design to see the overall structure and data flow.

a. State Machines

There are two ways to model a state machine in VHDL. The first is to use one process and define every state and output transition within that process. The second, which is the one chosen for the convolutional encoder model, is to separate the state transitions and outputs into two different processes referred to here as the "state" process and the "output" process. The state process has the clock and reset in the sensitivity list and defines only state transitions. The output process is either a concurrent selective signal assignment statement with the state as the selecting signal, or it is a concurrent PROCESS statement having only the state in the sensitivity list. This structure works well for both Mealy and Moore machines. The only difference is that the Mealy machine has signals as

23

Listing 4.1. Excerpt from SEQUENCER state machine source code (Appendix A).

```
s:PROCESS(clk, reset)
 BEGIN
  IF (reset = '0') THEN                    -- asynchronous reset.
    state <= state_0;
  ELSIF (clk'EVENT AND clk = '1') THEN     -- state machine transitions on
    CASE state IS                          -- rising clock edge.
      WHEN state_0 =>                      -- go to state_1 regardless of the inputs.
        state <= state_1;
      WHEN state_1 =>                      -- go to state_2 regardless of the inputs.
        state <= state_2;
      WHEN state_2 =>
        IF (n = "000" OR n = "001" OR n = "010") THEN
                                           -- if n is less than or equal to 2,
        state <= state_1;                  -- go to state_1.
        ELSE
          state <= state_3;                -- more than 2 bits/n-tuple.
        END IF;
      WHEN state_3 => .....
```

inputs to the selective signal assignment or PROCESS statement's sensitivity list, whereas the Moore machine has constant literal values. Listing 4.1 is an excerpt from the state process of the SEQUENCER block. The state transitions are a function only of $n(2:0)$, and a state transition is triggered only on a rising clock edge or an asynchronous reset. The complete VHDL source code for the SEQUENCER block and the other blocks is in Appendix A. Additionally, the VHDL code was written based upon the state diagrams described in the previous chapter. Refer to them if necessary to trace through the VHDL code.

b. Multiplexors

Listing 4.2 is an excerpt from the output process of the SEQUENCER block. As mentioned above, it describes the output signal transitions of the SEQUENCER state

machine, and it also serves as a multiplexor example. This VHDL construct is a selective

signal assignment, similar to the sequential CASE statement. It is a concurrent process

Listing 4.2. Excerpt from SEQUENCER source code (Appendix A).

```
-- mux structure that uses state flip-flops to select bits of "mod2_sums" for output
WITH state SELECT
serial <= mod2_sums(1) WHEN state_1,
          mod2_sums(2) WHEN state_2,
          mod2_sums(3) WHEN state_3,
          mod2_sums(4) WHEN state_4,
          mod2_sums(5) WHEN state_5,
          mod2_sums(6) WHEN state_6,
          '0' WHEN state_0;
```

executing in the same *simulation* time as all other concurrent statements and concurrent

PROCESS statements. The hardware representation is a multiplexor with the signal "state"

determining which input signal is assigned to the signal "serial". Thus, "state" is the

selection input, "mod2_sums(6:1)" are the inputs, and "serial" is the output.

c. Implicit Storage Elements

Listing 4.3 shows a portion of the DATAREG source code describing an

implicit storage register with a load enable and an asynchronous reset. Earlier in the code

Listing 4.3. Excerpt from DATAREG showing implicit storage register (Appendix A).

```
PROCESS (clk, reset)
BEGIN
  IF (reset = '0') THEN              -- asynchronous clear
    q <= "00000000";
  ELSIF (clk'EVENT AND clk = '0') THEN -- clock on falling edge.
    IF (load = '0') THEN             -- Q outputs get D inputs
      q <= d;                        -- only if "load" input is low.
    END IF;
  ELSE
  END IF;
END PROCESS;
```

25

(see Appendix A), BIT_VECTORs "d()" and "q()" were declared. The value of "d()" is assigned to "q()" only when the load enable signal is low and the clock is on a falling edge. If "reset" goes low then "q()" is assigned zeros regardless of the clock, "d()" or "load". The code describes an implicit storage register because if the conditional statements do not evaluate to true, "q()" is not assigned the value of "d()". Thus, it is implied that "q()" retains its old value and therefore is "stored". No component with specific ports is explicitly instantiated, yet the behavior is that of a register. Storage is implied any time an assignment statement is used inside a synchronization construct, which uses the 'EVENT attribute in the conditional part of an IF...THEN...ELSIF statement. Storage is also implied for assignment statements inside an incompletely specified conditional assignment statement (that is, an IF...THEN with no ELSE). Looking back at Listing 4.1, implied storage is an inherent part of state machines. (Harr, 1991, p. 149).

B. SIMULATION

The simulation procedure was quite simple. All the polynomials representing the **k** and **g** vectors were multiplied together as described above to obtain the rows of Table 1. Then simulations were run in the QuicksimII environment using all appropriate combinations of k and n $(n > k)$ in the STIMULUS block. The resulting "serial_out" waveforms in the Trace window were checked against the code sequences derived from Table 1. Figures 1 through 4 depict the output from the simulator for rates 2/3, 1/6, 1/2, and 3/5, respectively. The "serial_out" waveforms were used as the standard against which the FPGA implementation was checked.

Table 1 shows the code bits used to verify the simulation outputs. It contains the code bits for any code working on the test message pattern "1001110101" with connection vectors $g_1 = 215_8$, $g_2 = 251_8$, $g_3 = 242_8$, $g_4 = 236_8$, $g_5 = 223_8$ and $g_6 = 275_8$ and an 8-bit DATAREG. Each row comes from the product of the polynomial representations of **k** and

26

one of the connection vectors, **g**. Note that polynomials representing bit vectors are found by associating a power of X, starting with X^0, with each bit position. With the bit positions numbered 0 through 7 from left to right, the bit position becomes the exponent in the corresponding polynomial term if the bit is a '1'. If the bit is a '0', no term appears in the polynomial. To arrive at row $\mathbf{kg_1}$, the polynomial $(1+X^3+X^4+X^5+X^7+X^9)$, representing the test message pattern, is multiplied by $(1+X^4+X^5+X^7)$, representing the connection vector $\mathbf{g_1}$. Remembering that the partial products are modulo-2 added, the product is $(1+X^3+X^7+X^9+X^{13}+X^{16})$, which is the polynomial representing row $\mathbf{kg_1}$.

TABLE 4.1. CODE BIT PATTERNS FOR TEST MESSAGE PATTERN "1001110101".

	k MULTIPLES																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\mathbf{kg_1}$	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	1
$\mathbf{kg_2}$	1	0	1	1	0	0	1	0	1	1	1	1	1	1	1	0	1
$\mathbf{kg_3}$	1	0	1	1	1	0	0	0	0	1	1	0	0	1	0	1	0
$\mathbf{kg_4}$	1	0	0	0	0	0	0	1	1	0	1	0	0	0	1	1	0
$\mathbf{kg_5}$	1	0	0	0	1	1	0	1	1	0	1	0	0	1	1	1	1
$\mathbf{kg_6}$	1	0	1	0	0	1	0	1	1	0	1	1	1	1	0	0	1

The output code bit sequence of any code rate k/n is found by entering the table at column number k, reading down the column n rows, then repeating this procedure in columns $2k$, $3k$, and so on. Thus, for a rate 3/5 code, the first five code bits, "01100", are found in column 3, the second five, "00001", are found in column 6, and so on. The 3/5 code generates the pattern "01100 00001 01011 01000 01011". The 2/3 code generates "000 111 000 100 111 010 111 001".

C. STIMULUS

The STIMULUS block provides a serial test message pattern to the encoder. During development the test pattern was "1001110101 0000000" with the leftmost bit transmitted first. The trailing zeros are necessary to flush DATAREG. Since all of the bits in

DATAREG affect the output code bits, the coded stream is not complete until the last '1' has transited completely through the register. STIMULUS is written to repeatedly transmit the test message forever.

One of the great advantages of VHDL is that both the design and the test code is written in the same language. Therefore, the test block is thought of as just another hardware model with inputs and outputs. Thus, the design under test can provide inputs to the test block and the test block can respond with different test outputs as appropriate. In this case, STIMULUS takes as an input the signal "en" from the "IN_ENBLE" block. Recall that "en" allows k message bits into the encoder. STIMULUS provides input only as long as "en" is high, just as an actual system would behave if the encoder design was a component of the system. This block could have been written to have more of the behavior of the parent system, such as the ability to load REGFILE with code parameters, but the simulation emphasis was on checking that the convolutional encoding was correct. The VHDL code for STIMULUS was purposely kept quite simple to minimize debugging. A STIMULUS block with bugs obviously would cause incorrect results from the circuit under test.

Most of the STIMULUS code simply makes sure that the first bit of the test message is not sent in the same high "en" pulse as the final bit. When the final bit of the test pattern is sent, a flag is set. As long as the "en" pulse is active, the flag prevents STIMULUS from starting over at the beginning of the test pattern until a new "en" pulse has arrived. This guarantees that the first bit of the test message pattern is always the first bit sent within an active "en" pulse. For simulation purposes, this scheme synchronizes the test pattern to the encoder operation so that the beginnings of the repeated test patterns could be easily located in the QuickSimII output waveforms.

28

This chapter concludes the discussions on the high level behavioral design and simulation of the programmable convolutional encoder. The next chapter is the first of several that deal with specific hardware details associated with Field Programmable Gate Arrays (FPGAs) and how to translate the behavior covered above into hardware.

V. FIELD-PROGRAMMABLE GATE ARRAYS

This chapter describes FPGAs and the types that are available. Because it is important to have a detailed knowledge of the target FPGA architecture to get best performance, this chapter also provides a very detailed description of the line of FPGA made by Xilinx, Inc., called the Logic Cell Array (LCA). The Xilinx XC3064 LCA was used to implement the programmable convolutional encoder design.

A. INTRODUCTION

Field Programmable Gate arrays are standard, off-the-shelf VLSI devices whose functionality the user defines. They consist of a pattern of logic blocks surrounded by interconnection paths. There are four types available, shown in Figure 5.1.
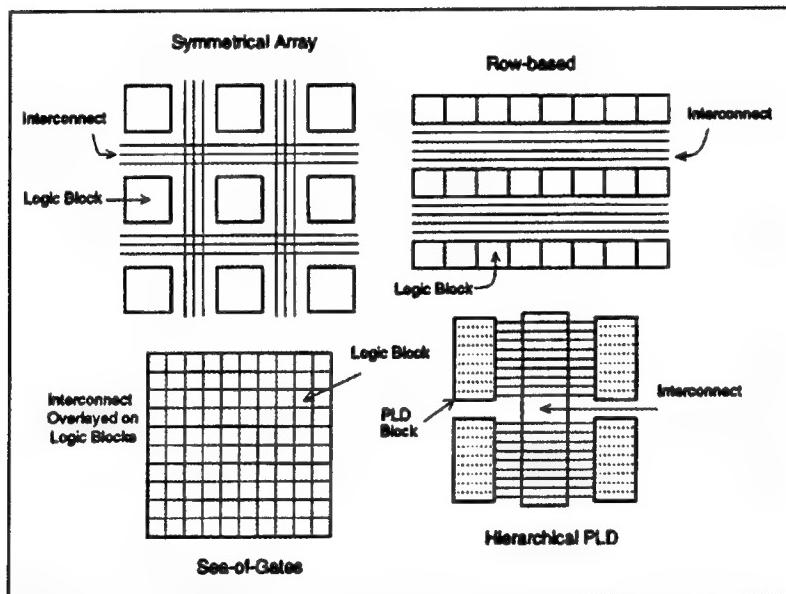

Figure 5.1. The four types of FPGA (Brown, 1992, p. 14)

Each logic block contains combinational circuitry such as multiplexors, look-up tables, or a PLD that the user programs to implement Boolean functions. The blocks also have flip-flops which can store either the output of the Boolean function or other signals routed into

the block but bypassing the logic structures. The interconnection resources consist of metal segments and programmable switches which route signals between the logic blocks. Each block implements a small piece of the overall design, and the interconnection resources connect all the pieces together into a complete digital design. The designer uses CAD software to generate a binary file from a schematic diagram or from a hardware description language and then downloads the file to the FPGA to configure the logic blocks and interconnection resources.

Because FPGAs can implement large digital circuits on a single chip, they offer huge advantages in system size, power consumption, and speed over systems built with SSI and MSI technology. They are commonplace in today's new electronic systems implementing random logic and application-specific functions. Most types are reprogrammable. This feature makes FPGAs ideal for prototyping new systems and for changing the structure of an existing system in the field.

The two most important benefits to using these devices are, first, convenience, and second, low cost. FPGAs provide inexpensive, instantly verifiable prototypes of complex digital circuits. As a system develops, the user can repeatedly change the design by downloading a new configuration program into the device. Not all FPGAs are reprogrammable, however. Some types, referred to as "one time programmable", are permanent once programmed and must be discarded if changes become necessary. The Xilinx device used in this thesis uses static RAM technology to set up the logic blocks and switching resources, and the user can reprogram it an unlimited number of times.

- The second major benefit is low cost. Other avenues to custom or semi-custom VLSI devices involve high non-recurring engineering (NRE) costs that are associated with tooling a commercial foundry to produce a device with the desired functionality. One such device is the Mask-Programmable Gate Array (MPGA). This device consists of rows and columns of

31

transistors that are connected according to the user's specifications. However, the foundry must produce the metal mask layers and deposit the metal interconnect onto the die. This is costly. These costs are usually in the tens of thousands of dollars and occur only once during production of the design. This cost translates to a per unit cost much higher than that of an FPGA for volumes less than about 1000 units (Brown, p. 4). Consequently, for low volume systems, FPGAs are used in the final systems as well as in the prototypes.

Despite the low cost and convenience offered by FPGAs, they have some limitations. The programmable switches in the routing paths introduce extra resistance and capacitance which would not be present in a custom chip. The additional RC time constants slow the signals traveling between the logic blocks causing FPGA designs to be significantly slower (up to several times slower (Brown, p. 6)) than other VLSI implementations. Another limitation is lower logic density. The programming circuitry and switches that give FPGAs their programmable nature occupy space on the die which otherwise would be dedicated to the design itself. FPGAs can be 8 to 12 times less dense than MPGAs manufactured in the same fabrication process (Brown, p. 6).

A consequence, but not necessarily a limitation, of the FPGA architecture is that special design techniques must be used to squeeze all of the available performance out of these devices. The typical academic procedure for designing state machines with states encoded as a binary sequence, for example, often is not the best approach. FPGA architectures tend to have a high proportion of flip-flops compared to the combinational circuitry that feeds the flip-flop inputs. Consequently, highly encoded state machines such as binary counters can require several logic blocks worth of next state decoding logic. Effectively, these types of state machines require logic blocks in series, and their performance suffers from the added propagation delay introduced by the extra interconnect. The problem can be avoided by using a different state encoding technique that uses more flip-flops. Using more flip-flops

tends to decrease the complexity of the next state decoding circuitry, reducing the number of logic blocks and the combinational delays. Design techniques which take advantage of one-hot state encoding and shift register structures like Linear Feedback Shift Registers (LFSRs), Johnson counters, and ring counters are well suited to FPGA architectures because they require relatively little combinational circuitry. (Knapp, Klein)

B. XILINX XC3064 ARCHITECTURE

The FPGA used for the convolutional encoder design is the XC3064 from Xilinx, Inc. Figure 5.2 shows the general structure of all Xilinx FPGAs, which Xilinx calls Logic Cell Arrays (LCA). The XC3064 consists of an 16 x 14 matrix of 224 Configurable Logic Blocks (CLBs) surrounded by 120 Input/Output Blocks (IOBs). The CLBs implement the logic design, and the IOBs provide an interface between the design and the package pins. Programmable interconnection channels run horizontally and vertically between the CLBs and around the CLB matrix. Static RAM cells control the programmable functions of the LCA.
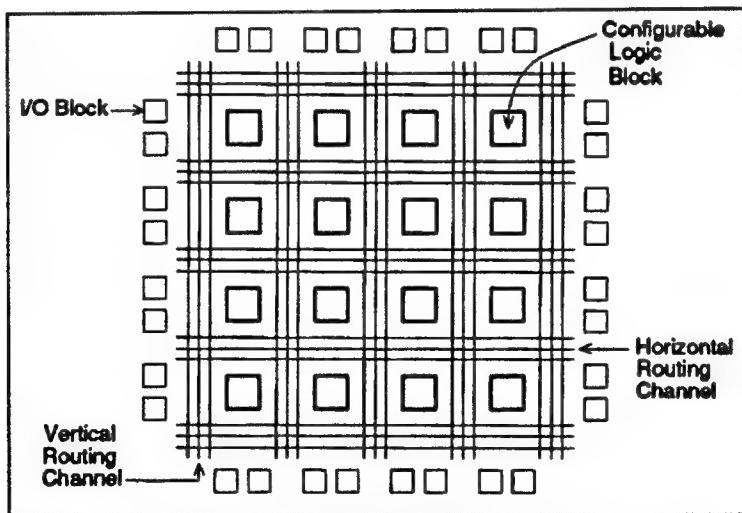

Figure 5.2. Xilinx LCA structure (Brown, 1992, p.22)

1. Configurable Logic Block

Figure 5.3 shows a CLB. Each CLB contains a 5-input look-up table (LUT), two flip-flops, and multiplexors to route signals between the flip-flops, the LUT, and the CLB inputs and outputs.

a. Multiplexors

The CLB contains two types of multiplexors. The first type is denoted by the traditional rectangular multiplexor symbol with the control line entering the bottom. It can either route the Q flip-flop output back to the D input, disabling the flip-flop, or it can route the F, G, or DATA IN (IN) signals to the D input. Its control line is either of the inputs ENABLE CLOCK (EC) or (ENABLE). Each flip-flop D input is fed by one of these multiplexors.

The second type of multiplexor is denoted by the trapezoidal symbol without a control line. This type controls the configuration of the CLB. The selection signals for these multiplexors come from static RAM cells that hold bits of the configuration program downloaded by the user. Since these bits do not change after the configuration program has been downloaded, the control lines are not shown. Two multiplexors select which signal feeds the D input of the flip-flops (F, G, DIN), and two more configure the X and Y CLB outputs as registered or combinational. The remaining three select the clock line (inverted or noninverted), the clock enable line (ENABLE CLOCK or (ENABLE)), and the effect of the reset line entering the CLB from the routing channels (DIRECT RESET or (INHIBIT)).

b. Look-up Table

The LUT has five inputs and two outputs. It is a 32x1 table which can implement one function of five variables or two functions of four variables. There are seven physical inputs to the LUT: CLB inputs A, B, C, D, E, and feedback signals QX and QY. However, a maximum of five of these seven are used to implement Boolean functions. For a

34

5-variable function, three are A, D, and E. The fourth is any one of B, QX, or QY, while the fifth variable is any one of C, QX, or QY. In this case, the LUT outputs, F and G, are identical. See Figure 5.4b.
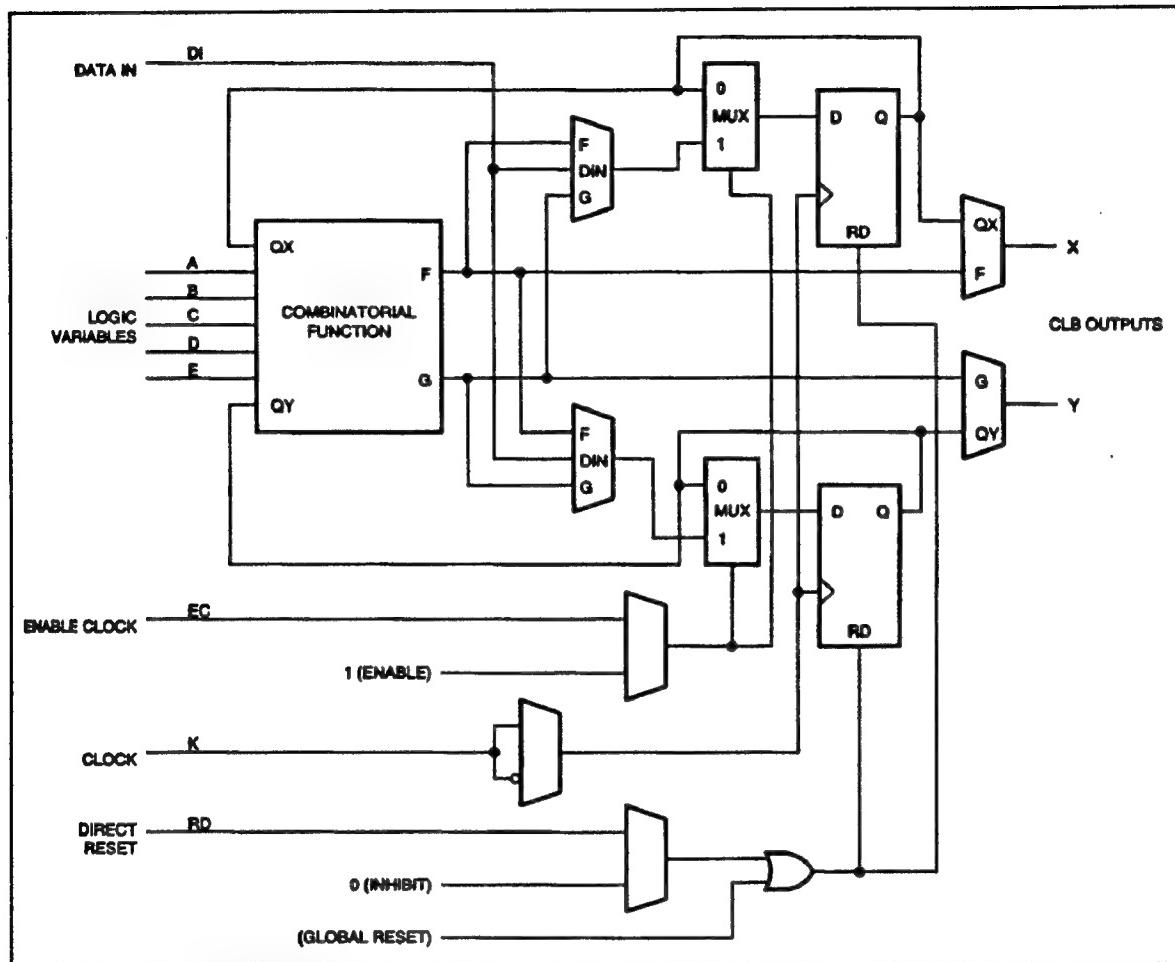


Figure 5.3. Configurable Logic Block (Xilinx, 1994, p.2-109)

Similarly, the inputs to the two 4-variable functions are groupings of the seven physical inputs. One variable is A which must be common to both functions. For both functions one input is either B, QX, or QY, and another input is either C, QX, or QY. The fourth is either D or E. The outputs F and G are independent. See Figure 5.4a.

Some 6- and 7-variable functions can be implemented, but because of the physical structure of the LCA, these functions must be in the form:

$$F = f_1(A,L,M,D) \cdot !E + f_2(A,L,M,D) \cdot E$$

where E is the select input of a 2-to-1 multiplexor (the exclamation point implies Boolean negation), f_1 and f_2 are two 4-variable functions feeding the data inputs of the multiplexor, and L and M can each be B, C, QX, or QY. A further constraint is that at least two of the inputs to f_1 and f_2 (inputs A and D) must be common to both functions. As with the five-variable case, LUT outputs F and G are identical. See Figure 5.4c.

c. Storage Elements

Each CLB contains two D-type flip-flops. User-programmed multiplexors, mentioned above, select the source of each D-input from either the flip-flop's own Q output, the F or G LUT outputs, or the DATA IN (DI) input which bypasses the LUT. If the CLB is configured for registered outputs, one Q-output becomes the X CLB output, and the other becomes the Y CLB output. The Q-outputs also go to the QX and QY inputs of the LUT. The flip-flops are clocked by the invertible CLOCK (K) input, and they are asynchronously reset by either DIRECT RESET (RD) or GLOBAL RESET.

2. Input/Output Block

The IOBs surround the 8-by-8 array of CLBs and provide an interface to the package pins. Each pin can be used as an input to the device or as an output. Figure 5.5 shows an IOB. Each block contains an output D-type flip-flop that can provide a registered signal to a pin configured as an output pin. In addition, the IOBs have an input storage element which can be set up as either a D-type flip-flop or as a D-type latch to store signals from pins configured as inputs. The asynchronous resets of both storage elements connect to the LCA's global reset line, GLOBAL RESET. Their clock inputs connect to either of

36
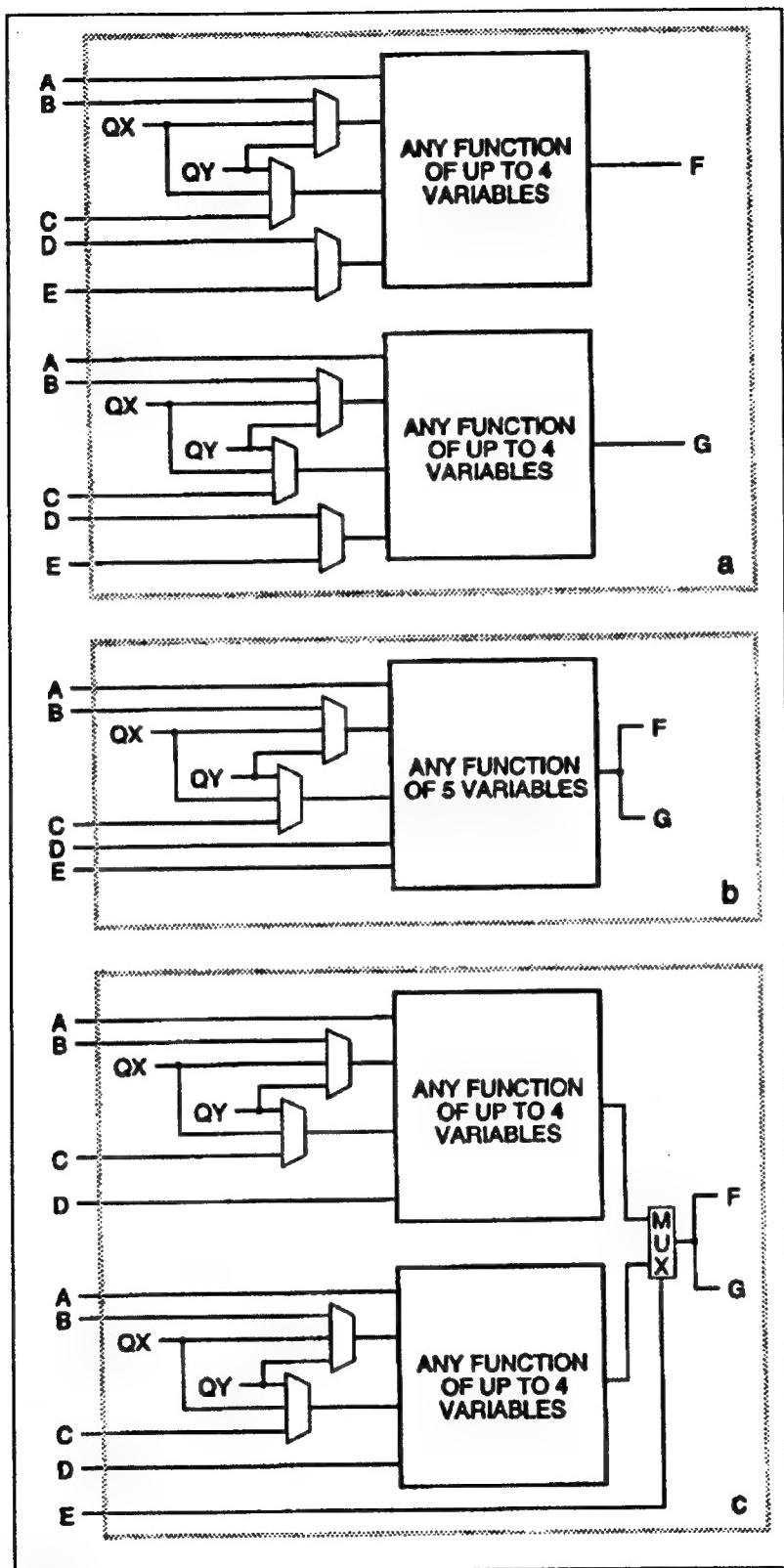
Figure 5.4. Look-up Table Usage (Xilinx, 1994, p. 2-110)

two clock lines, CK1 and CK2, which lie on the edges of the LCA die. Each clock is invertible for the die as a whole, but not for any individual storage element.
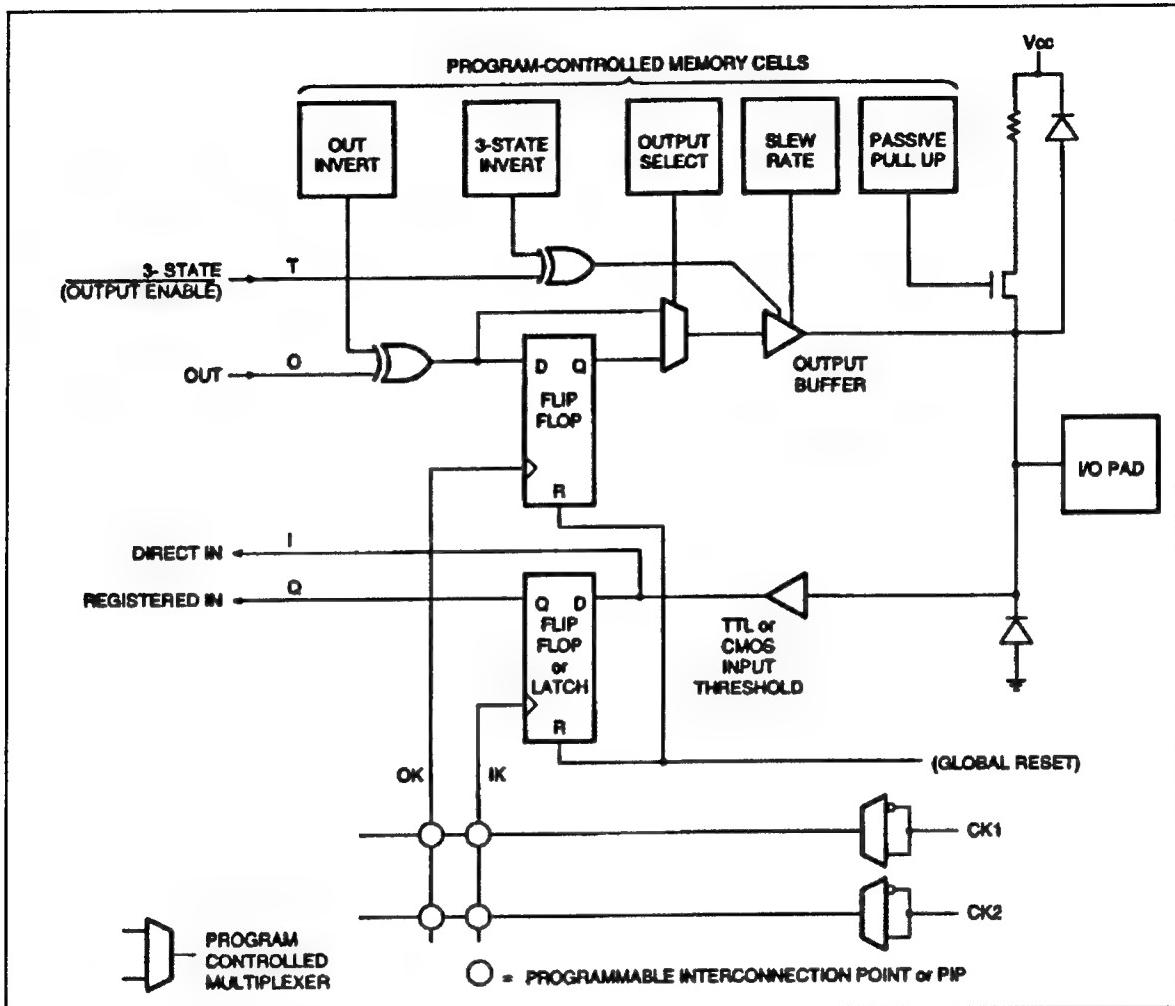
Figure 5.5. Input/Output Block (Xilinx, 1994, p.2-107)

If a particular I/O pin functions as an output, its signal, OUT (O), comes through a programmable 3-state output buffer from a 2-to-1 multiplexor which selects the registered or combinational version of the signal. An XOR gate, one of whose inputs connects to a program-controlled memory cell, can invert the signal before it arrives at the flip-flop. The active logic level of the buffer control, 3-STATE (T), is invertible in a similar manner.

When an I/O pin is an input, the signal passes through an input buffer whose input thresholds are programmed for TTL or CMOS levels. This is a global feature of the die, not a block-by-block programmable feature. The signal then feeds the input storage element for latched inputs, REGISTERED IN (Q), and for combinational inputs, DIRECT IN (I), bypasses the element for direct input to the interconnection resources.

3. Configuration Memory

The control of the multiplexors, XOR gates, 3-state output buffer, and pull-ups, and switching resources comes from a configuration program which loads from external memory into the LCA on power-up or on the user's command. The program loads an array of static memory cells that are distributed throughout the LCA. The outputs of these cells configure all of the programmable features.

4. Programmable Interconnect

The programmable interconnect resources consist of three types of interconnection between CLBs and IOBs: (1) General Purpose, (2) Direct, and (3) Longlines. These structures connect the blocks on the LCA to implement the user's digital design.

a. General Purpose Interconnect

Five general purpose interconnect metal lines run the length and width of each CLB or IOB. At each corner, a switching matrix provides the interconnectivity between the four sets of five lines meeting at that particular junction. Each line can connect to between four and six other lines, depending on which line carries the input signal. Figure 5.6 shows the various configurations of a switching matrix.

b. Direct Interconnect

Direct interconnection allows CLBs to connect their outputs directly to neighboring CLBs or IOBs, bypassing the general interconnect switching matrices and lines. This method presents the least delay to signals traveling between adjacent blocks. The X

39

CLB output can connect to the B input of the CLB to its right and to the C input of the one to its left. The Y CLB output can connect to the D input of the CLB above and to the A input of the CLB below. The CLBs neighboring IOBs connect to the two closest IOBs. One CLB output goes to one IOB, and one input comes from the other IOB.
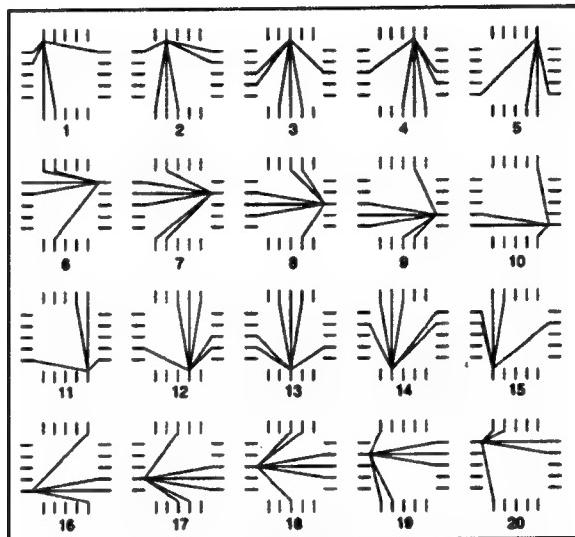
Figure 5.6. Switching Matrix Configurations
(Xilinx, 1994, p. 2-113)

c. Longlines

Longlines run the width and height of the interconnect area, bypassing the general interconnect switching matrices. Every column of the interconnect area has three longlines and every row has two. Two more run along the outer sets of switching matrices. Longlines carry signals which must travel a long distance or which require minimal skew.

The next chapter covers design methods that optimizes hardware performance by taking advantage of some of the characteristics of the FPGA architecture.

VI. STATE ASSIGNMENT

Using the SEQUENCER block as an example, this chapter compares the one-hot state assignment technique to the standard binary state assignment technique. It also describes how to use redundant states to help take advantage of the LUT based architecture of the Xilinx LCA.

A. ONE-HOT vs. BINARY

One-hot state assignment is a scheme whereby each state in a state machine is represented by one and only one active flip-flop. There are at least as many flip-flops in the state machine as there are states. Because each state is represented by only one flip-flop, no state decoding logic is necessary. Consequently, the one-hot state assignment reduces the next-state decoding logic because the next state of the machine is determined by the input and one active flip-flop. Overall, the complete circuit may have more logic than a binary encoded machine, but, on a per flip-flop basis, the simplified next-state logic replaces the deeper, slower, high fan-in logic of a binary encoded machine, thus decreasing logic delay between state transitions and enhancing speed.

Another benefit from the one-hot assignment is the ability to break a state with deep input logic into redundant states with simplified input logic. The transition equation for each redundant state is composed of small groups of product terms which were in the equation for the original state. This will be demonstrated later in this chapter.

The one-hot assignment is not always the best choice, however. As the number of states increases, the number of flip-flops increases one for one, whereas the number of flip-flops increases with $\log_2(S)$ (where S is the number of states) for binary assignment. For small state machines with few inputs and simple next-state logic, binary encoding might be the best choice because the number of flip-flops can be conserved.

41

For example, the Xilinx XC3000 family of devices uses a 5-input look-up table (LUT) to implement combinational logic. If a modulo-32 counter is needed in a design, and it requires no control lines besides an asynchronous reset, which is handled outside the LUT, then a binary state assignment would be satisfactory because the only inputs to the LUTs would be the five present state outputs of the flip-flops. This approach would also conserve flip-flops.

On the other hand, if only one control line is needed, perhaps an "enable", then one input of each LUT would be consumed for the control line, leaving only four for present state inputs. In this case, assuming the binary state assignment, a modulo-16 counter would be the largest counter possible without introducing an extra LUT (in another CLB) for each flip-flop. Therefore, every flip-flop would require two levels of CLBs to implement the modulo-32 counter. The additional delay between the CLBs would cut the counter's speed significantly. Under this condition, the binary state assignment would not be appropriate.

Another potential pitfall is that the number of invalid states in a one-hot assignment far outweighs the number of valid ones. A 5-state state machine requires five flip-flops if a one-hot state assignment is used, but there are 32 possible states associated with five flip-flops. Therefore, this relatively simple state machine would have 27 invalid states! The extra logic required to account for all or most of the illegal states could create longer signal paths and significantly slow down the state machine erasing the benefits of the one-hot state assignment. Thus, the one-hot assignment delivers simplicity and speed for the cost of lower reliability and inefficient usage of flip-flops.

The designer must be intimately familiar not only with the details of the design itself, but also with the target technology, which dictates the appropriate logic structures that give the best performance. Thus, the choice of state assignments is dependent upon the state machine itself and the technology implementing it. In the above example, a counter based on

a Linear Feedback Shift Register (LFSR) is the best solution because it requires only five flip-flops and little combinational circuitry to implement, making it appropriate for LUT-based FPGA technology.

B. LUT IMPLEMENTATION

The 5-input LUT implements the logic to the D-input of each flip-flop in the CLBs of the Xilinx 3064 LCA that was used to implement the encoder design. The actual structure of the combinational circuit is not a concern because the LUT has a constant delay across it regardless of the logic function it realizes (Xilinx, 1994, p. 2-111). The main concern is whether the LUT has enough inputs to accommodate the number of variables in the logic function. For a 5-variable function, the LUT must have five available inputs. The LUT acts as a 32 X 1 RAM whose 5 address lines are the five inputs of the logic function, and whose 1-bit outputs are the active or inactive result of each of the 32 possible product terms. To select state assignments for the state machines in the encoder design, each state machine was studied to determine the number of inputs necessary for the next state decoding logic for each flip-flop. The goal is to keep the number of inputs below five so that the complete decoding function for each state flip-flop is contained in the flip-flop's companion LUT. The SEQUENCER block is used here to illustrate the method and to compare to a binary assignment.

Table 6.1 shows the state table for the SEQUENCER block. The state diagram is shown in Figure 6.1. Table 6.2 lists the state transition equations and the required number of inputs to the LUT for each flip-flop. For example, s3 is the next one-hot state if SEQUENCER is in state 2 and the "n(2:0)" input is a binary pattern other than "010". As Table 6.2 shows, the fan-in to each one-hot state flip-flop except for s1 is less than five inputs, suggesting that the next state logic for those flip-flops can be completely contained in their respective

LUTs. State s1 must be split into several redundant states to simplify   fan-in logic. This matter is dealt with in Section C.

Table 6.3 shows the state transition equations for a state machine with a binary state

TABLE 6.1: STATE TABLE FOR SEQUENCER BLOCK

STATE	INPUT n(2:0)								OUTPUT
	000	001	010	011	100	101	110	111	
s0	s1	s1	s1	s1	s1	s1	s1	s1	0
s1	s2	s2	s2	s2	s2	s2	s2	s2	m(1)
s2	s3	s3	s1	s3	s3	s3	s3	s3	m(2)
s3	s4	s4	s4	s1	s4	s4	s4	s4	m(3)
s4	s5	s5	s5	s5	s1	s5	s5	s5	m(4)
s5	s6	s6	s6	s6	s6	s1	s6	s6	m(5)
s6	s1	s1	s1	s1	s1	s1	s1	s1	m(6)

assignment. The state assignment is: s0 = "000", s1 = "001", s2 = "010", s3 = "011", s4 = "100", s5 = "101", and s6 = "110". D2, D1, and D0 are the inputs to the state flip-flops, and they represent the next state of the machine. Note that in this table, s0 through s6 represent the 3-bit present state, whereas they represent a 1-bit present state in the one-hot

TABLE 6.2: STATE TRANSITION EQUATIONS (ONE-HOT)

STATE TRANSITION EQUATIONS	# LUT INPUTS
D0 = reset	0
D1 = s0 + s2·(010) + s3·(011) + s4·(100) + s5·(101) + s6	9
D2 = s1	1
D3 = s2·(!010)	4
D4 = s3·(!011)	4
D5 = s4·(!100)	4
D6 = s5·(!101)	4

assignment (Table 6.2). For example, flip-flop input D2 would be asserted high if the present state is s3 with "n(2:0)" = !("011"), if the present state is s4 with "n(2:0)" = !("100"), or if the present state is s5 with "n(2:0)" = !("101"). The exclamation

Figure 6.1. State diagram for SEQUENCER block.

point represents Boolean negation. Thus, the next state logic input for D2 consists of the 3-bit encoded state and the 3-bit input, n(2..0), for a total of six inputs to the LUT. Indeed, all three flip-flops for the binary state assignment require six or seven inputs. Since theBoolean equations for the D inputs cannot be placed in the proper form for a single LUT to implement as a 6- or 7-variable function (see Chapter V), more than one LUT is needed for every flip-flop D-input.

TABLE 6.3: STATE TRANSITION EQUATIONS (BINARY)

STATE TRANSITION EQUATIONS	# LUT INPUTS
D2 = s3·(!011) + s4·(!100) + s5·(!101)	6
D1 = s1 + s2·(!010) + s5·(!101)	6
D0 = reset + s1 + s3·(!011) + s5·(!101)	6

C. EXPLOITING REDUNDANT STATES

The one-hot technique is attractive for the SEQUENCER state machine because, as Table 6.2 shows, all states except s1 require fewer than five inputs and therefore only one LUT. State s1, as mentioned earlier, can be broken up into several redundant states: s1', s1", and s1'''. All three states yield the same output and proceed to the same next state as the original s1 under the same conditions that allowed the original transition. In this machine, s1 proceeds to only one state, s2, regardless of the input values. Therefore, s2 is the next state for all of the primed s1's. Table 6.4 shows the new state table and Figure 6.2

TABLE 6.4: STATE TABLE WITH REDUNDANT STATES (ONE-HOT)

STATE	INPUT n(2:0)								OUTPUT
	000	001	010	011	100	101	110	111	
s0	s1'	s1'	s1'	s1'	s1'	s1'	s1'	s1'	0
s1'	s2	s2	s2	s2	s2	s2	s2	s2	m(1)
s1"	s2	s2	s2	s2	s2	s2	s2	s2	m(1)
s1'"	s2	s2	s2	s2	s2	s2	s2	s2	m(1)
s2	s3	s3	s1"	s3	s3	s3	s3	s3	m(2)
s3	s4	s4	s4	s1"	s4	s4	s4	s4	m(3)
s4	s5	s5	s5	s5	s1'"	s5	s5	s5	m(4)
s5	s6	s6	s6	s6	s6	s1'"	s6	s6	m(5)
s6	s1'	s1'	s1'	s1'	s1'	s1'	s1'	s1'	m(6)

shows the new state diagram. Table 6.5 shows the new transition equations. The equations for s1', s1", and s1'" were previously part of the transition equation for s1. Groups of

46

product terms have been broken out and assigned to the new states of s1', s1", and s1"'. Now the next state decoding logic for each flip-flop (and each state) is simple enough to reside in a single LUT. Speed can be maximized.

Note that splitting a state in the binary state assignment would accomplish nothing because the states are encoded. Six LUT inputs would still be required to distinguish all of the states: three for the state machine inputs and three for the encoded states. In general,
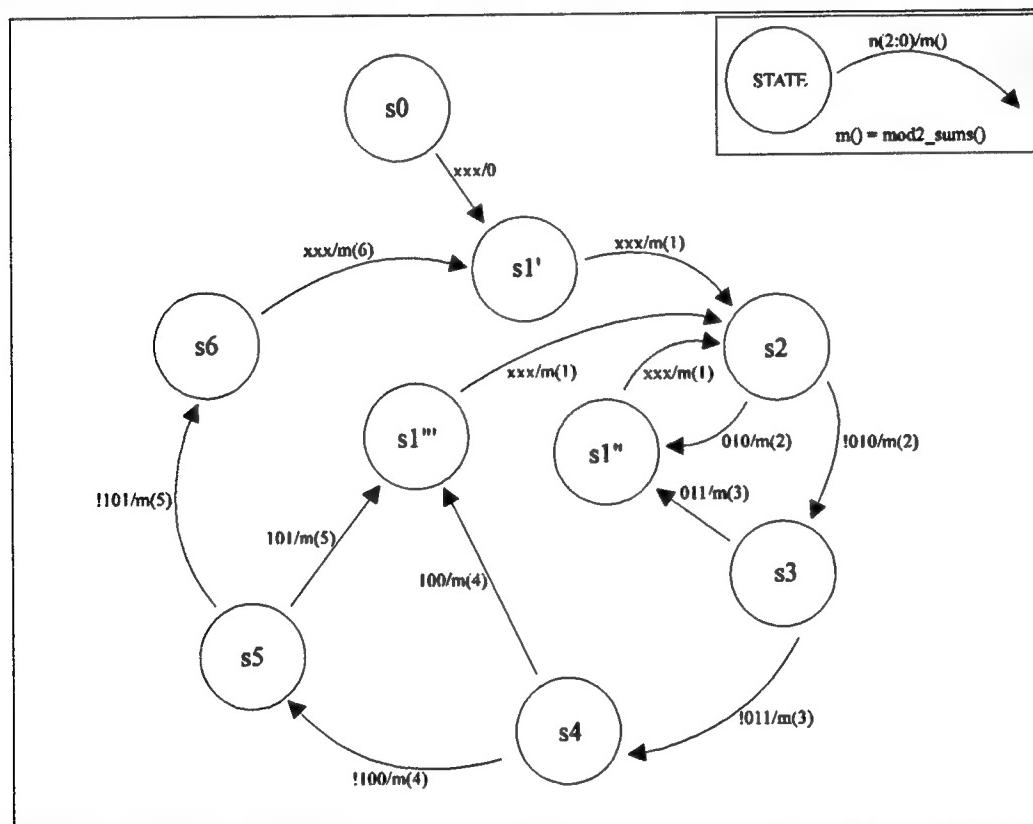
Figure 6.2. State diagram for one-hot assignment and redundant states.

with the one-hot state assignment, any complicated next-state logic can be broken down into a set of less complex circuits whose outputs are assigned to redundant states. There are fewer inputs required to activate these redundant states than for the original state so that all the combinational logic for each flip-flop may reside in the companion LUT.

All state machines in the encoder were synthesized with a one-hot state assignment after using this analysis on each machine. The handshaking state machine, HANDSHAK, is the

TABLE 6.5: STATE TRANSITION EQUATIONS WITH
REDUNDANT STATES (ONE-HOT)

TRANSITION EQUATIONS	# LUT INPUTS
D0 = reset	0
D1' = s6	1
D1" = s2·(010) + s3·(011)	5
D1"' = s4·(100) + s5·(101)	5
D2 = s1' + s1" + s1"'	3
D3 = s2·(!010)	4
D4 = s3·(!011)	4
D5 = s4·(!100)	4
D6 = s5·(!101)	4

only one appropriate for binary encoding because it has only three states. In fact, it fits into a single CLB.

The next chapter explains the process of adding pipeline registers to the encoder design and implementing it in the Xilinx XC3064PG138-100 LCA.

VII. FPGA IMPLEMENTATION

This chapter describes the sequence of events leading to an LCA implementation of the programmable convolutional encoder. It discusses the Xilinx CAD programs, and some minor differences between the hardware and the VHDL model. The chapter also covers the addition of pipelining registers, and offers comments on the use of the Mentor Graphics Autologic tool and back annotation into VHDL.

A. OVERVIEW

All state machine circuitry was derived from the state diagrams in Appendix C. These diagrams are the result of the same analysis procedure described for the SEQUENCER block in the last chapter. The circuitry for the remaining blocks of the design were produced directly from the VHDL source code by hand. Originally, this phase was to be done by the Autologic tool, but that tool was not useful for reasons outlined later in the chapter.

The design was implemented as closely as possible according to what is dictated by the VHDL source code. The only difference between the LCA and VHDL versions of the design is the behavior of the global reset line, "reset", and the latency that occurs as a consequence of pipelining. In the LCA version, "reset" is active high because the asynchronous reset of the individual flip-flops in the CLBs (input RD) are active high. Refer to Figure 5.3. Because all of the state machines have a reset state with one state flip-flop high, and because none of the flip-flops have a preset input, one input of the LUT feeding the high flip-flop must be used as an OR gate to force the flip-flop high on the clock edge following the activation of "reset". Therefore, "reset" is synchronous. Despite the fact that "reset" must be synchronous because of one preset flip-flop in the reset state of each state machine, the other state flip-flops are reset using the asynchronous RD inputs to avoid

49

wasting an LUT input for resetting. Table 7.1 lists the convolutional codes possible with this design.

B. IMPLEMENTATION FLOW

1. Schematic Capture

All schematic capture and Xilinx related development of the encoder design was done in the Mentor Graphics version 7.0 environment because this version is the only one for which the Xilinx macro libraries are installed. Despite the fact that all high level modeling was done in Mentor Graphics version 8.2, inputting schematics in the older version was not a problem as one might think. There were no gate level schematics done in the newer version, so there was no incompatibility problem with schematics being translated to the older version. Shifting to the older CAD system came at a convenient break in the implementation flow where schematics were manually derived from the VHDL code. After schematics were translated from VHDL, they were entered with LCA_NETED, the schematic capture program in Mentor Graphics, version 7.0.

2. Functional Verification

To verify functional operation, a TESTBENCH schematic was generated which incorporated the encoder block and a test circuit. This TESTBENCH is similar to the TESTBENCH concept used in VHDL modeling where a stimulus file interacts with the circuit under test while outputs and test points are monitored. The test circuit in this graphical TESTBENCH provides the same test message to the encoder as the VHDL STIMULUS file used in the high level model. By using the same test pattern, the output of the LCA implementation was easily compared to a known correct output produced from the VHDL model.

The test circuit only provided the test pattern, however, because it needs to take "en" from the IN_ENBLE block as an input, just as the STIMULUS block did in the high

TABLE 7.1. CONVOLUTIONAL CODES.
(Proakis, 1989, pp. 466 - 471)

Rate	L	Connection Vectors (octal)
1/2	3	5, 7
	4	15, 17
	5	23, 35
	6	53, 75
	7	133, 171
	8	247, 371
1/3	3	5, 7, 7
	4	13, 15, 17
	5	25, 33, 37
	6	47, 53, 75
	7	133, 145, 175
	8	225, 331, 367
1/4	3	5, 7, 7, 7
	4	13, 15, 15, 17
	5	25, 27, 33, 37
	6	53, 67, 71, 75
	7	135, 135, 147, 163
	8	235, 275, 313, 357
1/5	3	7, 7, 7, 5, 5
	4	17, 17, 13, 15, 15
	5	37, 27, 33, 25, 35
	6	75, 71, 73, 65, 57
	7	175, 131, 135, 135, 147
	8	257, 233, 323, 271, 357
1/6	3	7, 7, 7, 7, 5, 5
	4	17, 17, 13, 13, 15, 15
	5	37, 35, 27, 33, 25, 35
	6	73, 75, 55, 65, 47, 57
	7	173, 151, 135, 135, 163, 137
	8	253, 375, 331, 235, 313, 357
2/3	2	17, 06, 15
	3	27, 75, 72
	4	236, 155, 337
2/5	2	17, 07, 11, 12, 04
	3	27, 71, 52, 65, 57
	4	247, 366, 171, 266, 373
3/4	2	13, 25, 61, 47
3/5	2	35, 23, 75, 61, 47
4/5	2	237, 274, 156, 255, 337

51

level model. The version 7.0 environment does not have a VHDL compiler, so the test circuit and a MISL file were used instead. MISL files cannot take inputs, so the test circuit provides the test pattern because it must react to the response of the encoder. The MISL file provides connection vectors. The clock period, k(2:0), n(2:0), and reset were controlled from the command line during each simulation run. Outputs were checked against the same four code rates used in the high level model: 1/2, 2/3, 3/5, and 1/6. The program LCA_EXPAND_SIM was run on the schematic to convert it to a format compatible with the Mentor Graphics QuickSim simulator.

3. LCA Implementation

To progress from a schematic of the design in the Mentor Graphics environment to an LCA implementation, the following CAD programs were run in the order given.

a. *LCA_EXPAND and EREL2XNF*

LCA_EXPAND reformats the schematic into a format appropriate for input to EREL2XNF which outputs a Xilinx Netlist Format (XNF). (Messa, 1991, p. 67) The XNF file is a standard format used by the Xilinx Development System. Designs described with Boolean equations, schematics, or hardware description languages are converted to XNF files before further processing. (Lautzenheiser, 1989, p. 2)

b. *XNFMAP*

XNFMAP maps the logic defined by the XNF file to CLBs and IOBs and removes unnecessary logic. It places the resulting logic partitioning into a MAP file, which is the input to MAP2LCA, and creates a cross-reference report file (CRF) which contains a summary of LCA resource usage and cross-references between original logic elements and LCA design elements. (Xilinx, 1991, p. 13-1)

52

c. MAP2LCA

The MAP2LCA program uses the data in the MAP file to partition the design into a particular Xilinx LCA, in this case an XC3064PG132-100, and places the results into a Logic Cell Array (LCA) file. It also creates a constraints file (SCP) that contains the initial placement of the design and lists placement and routing constraints specified in the schematic. Lastly, it abbreviates full hierarchical path names of signal and symbols and lists them in an AKA file. (Xilinx, 1991, pp. 7-1 - 7-3)

4. CLB Placement and Routing

a. Automatic Place and Route

The Automatic Place and Route (APR) program takes an LCA file as input and uses the popular optimization algorithm called Simulated Annealing to generate an optimal placement of CLBs in the LCA architecture to minimize delays. Documentation is written to a report file (.rpt) and the routed design is written to another LCA file. The input LCA file may already have placement and routing information from a previous APR run. Using the correct command line option with APR allows the user to add more features to an already routed design. This practice Xilinx calls "incremental design". The APR program has many options that provide the user with varying degrees of control over the APR process. The user can even tell APR exactly where to place CLBs that contain particular parts of the design. The user exercises control over APR with Constraint Files.   (Xilinx, 1991, pp. 2-1 - 2-11)

b. Constraint Files

It is impossible for the APR program to know which signals of the design are the most critical simply by looking at the input LCA file. If it gets no outside advice from the designer, it randomly decides where to place CLBs and which signals to route in the faster routing resources on the LCA. If the critical path ends up smaller than the clock period, this

is satisfactory. Otherwise, the design will be too slow. The higher the performance needed from the LCA, the more help is needed from the user.

To give the APR program guidance, a User Constraint file is used. The entries in this file override any guidance derived solely from the schematics themselves. Using the constraints file, the designer can give APR implementation hints such as where to place certain CLBs, which type of routing resources to use for the timing-critical signals, which blocks and nets to freeze before placement and routing of additional circuitry, and which areas of the LCA to leave open. Taking full advantage of constraint files requires a very detailed knowledge of both the LCA architecture and the capabilities of the Xilinx software package. Xilinx has issued many Application Notes about its products. They should be studied carefully to realize high performance designs. (Xilinx, 1991, p. 2-12)

Constructing the constraints file was a very tedious process. Because the XNFMAP program eliminates some unnecessary logic and attaches cryptic names to all the nets, it is necessary to study the cross-reference report (CRF) file along with the schematic diagram to discern which signal is which. The new net names were used in the constraints file. The solution to this tedium is to name all the critical nets as the schematic diagrams are entered. The names are retained throughout the implementation flow.

5. Functional Verification of Back-Annotated Design

a. LCA2XNF

If a placed and routed LCA file is translated to an XNF file with the LCA2XNF program, then the XNF file contains worst-case block and net delays. In that case, the XNF file is called *back-annotated* and can be simulated in QuickSim to verify timing requirements. (Xilinx, 1991, p. 4-1) The same TESTBENCH was used as for the functional verification of the design before back annotation. The TESTBENCH for both versions (before and after annotation) and an output waveform are in Appendix F.

54

b. LCA_TIMING

This program takes a placed and routed LCA file and produces a new SIMSHEET which QuickSim uses for input. (Messa, 1991 p. 67)

C. PIPELINING

There were three places where propagation delays needed improvement: (1) in the 8-to-1 multiplexors of DATAREG, (2) in GENERATOR, and (3) in the SEQUENCER output. The Xilinx development system divides an 8-to-1 multiplexor into two levels of CLBs that contribute two block delays plus routing delays. Message bits shift into SHIFTREG and sit there until the "load" input to DATAREG is active (low), and then they are shifted in parallel into DATAREG on a falling clock edge. Since GENERATOR is combinational, these changes at the output of DATAREG travel through GENERATOR to the input of SEQUENCER. At the following rising edge, SEQUENCER updates its state machine. Thus, only one half of a clock cycle was available for the new logic levels to get through GENERATOR. This path consisted of four levels of combinational CLBs between two registered ones making it the longest combinational path, or the critical path of the design.

To eliminate DATAREG's input multiplexors and to improve GENERATOR's combinational delay, SHIFTREG and DATAREG were replaced by a serial-to-parallel shift register with a clock enable input, and pipeline registers with a clock enable were added to GENERATOR. With the same control signals, the behavior was preserved. The "load" signal which formerly was an input to DATAREG, is now the "calc" (short for "calculate modulo-2 sums") input to GENERATORDFF. The "load" signal enables the pipeline registers to save a partial sum. The pipeline registers in GENERATORDFF now hold the input to SEQUENCER while the shift register performs both DATAREG's and SHIFTREG's former functions. The shift register now receives *k*-tuples and holds them as

55

input for GENERATORDFF. The two levels of combinational delay due to the 8-to-1 multiplexors is gone, and the pipeline registers added to GENERATOR were available as flip-flops in the CLBs that realized GENERATOR initially. Therefore, no additional routing delay was added by incorporating the pipeline registers. The schematic of the new pipelined version of GENERATOR, called GENERATORDFF, is in Appendix E. Pipelining also improved the throughput of SEQUENCER's output stage. The schematic of the new SEQUENCER, dubbed PIPESEQUENCER, is in Appendix E.

1. Pipeline Register Placement

It is fairly simple to look at the schematic diagram of a block and see how the Xilinx Development System will partition the circuitry into CLBs. This was done to find appropriate locations for pipeline registers in GENERATORDFF and PIPESEQUENCER. Look at the GENERATOR block as an example. Examining one AND/XOR tree and remembering that each CLB look-up table can have four or five inputs, it is plain that the partitioning will occur as in Figure 7.1. Each tree consists of five CLBs arranged in two levels. The first level has four CLBs (one of which is delineated by the box) that feed into the second level which has only one CLB (also in a box). This was verified in the Xilinx XACT tool, which allows the user to navigate through the LCA to see how APR routed signals and configured CLBs. The pipeline registers are added to the schematic at the point shown in the figure so the next APR run will produce the same CLB partitioning but with the CLB outputs registered. Thus, pipeline registers are added without incurring any additional delay since the flip-flops reside in the CLBs anyway and just need to be wired in. The GENERATORDFF schematic shows the register placement. The "mod2_sums(6:1)" lines also go through a pipeline register which is in the PIPESEQUENCER schematic.

56
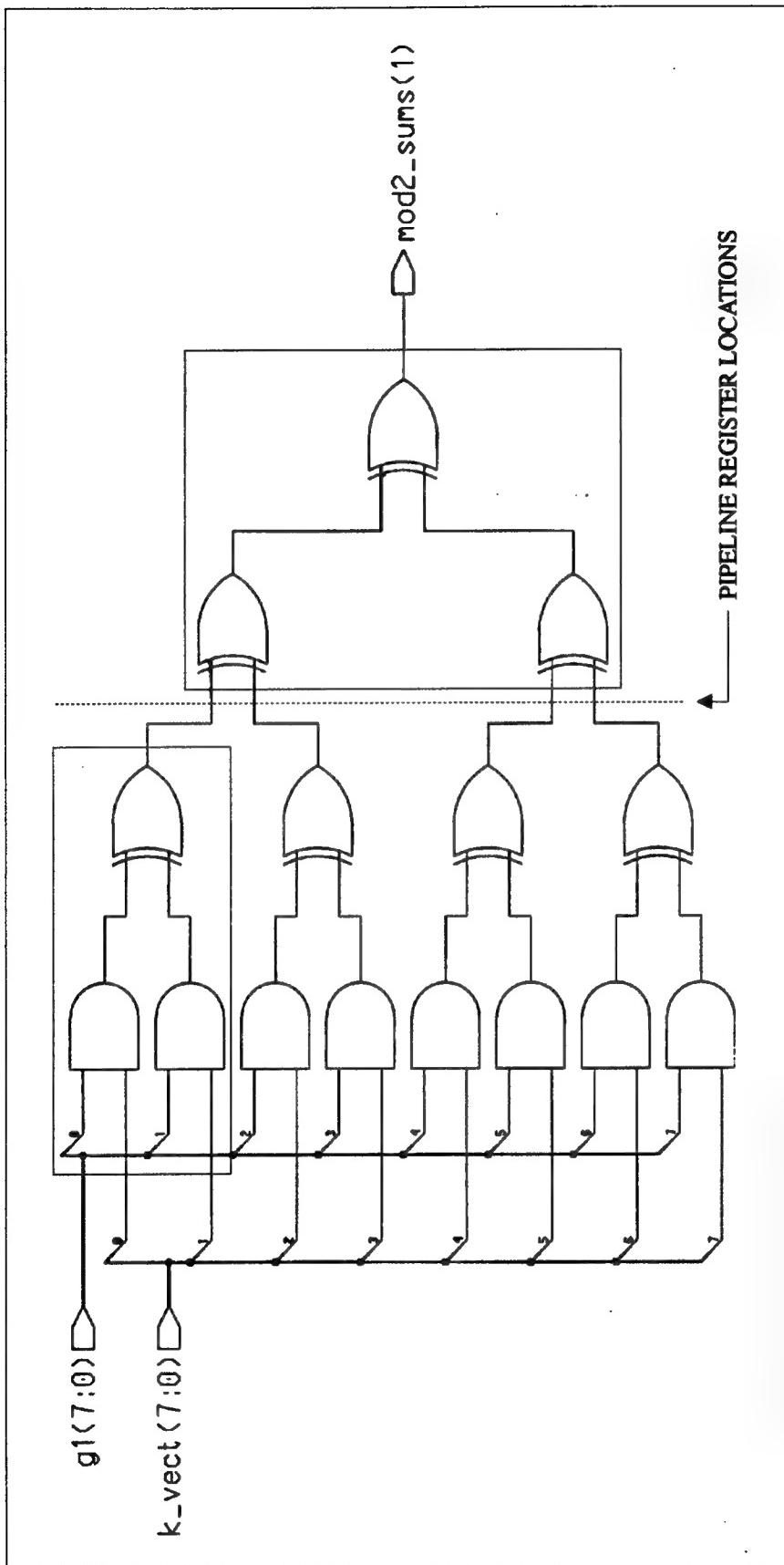
Figure 7.X. Determining pipelining register placement.

D. DESIGN PERFORMANCE

1. Propagation Delay Estimation

Signal paths in LCA devices generally start at the output of a flip-flop and travel through one or more levels of combinational CLBs to the input of another flip-flop. Therefore, the signals traveling the path are subject to the clock-to-output delay of the source flip-flop, routing delay between CLBs, combinational CLB propagation delay, and set-up time of the destination flip-flop. For estimation purposes, assume 18ns, total, for clock-to-output delay of the source registered CLB plus the set-up time for the destination registered CLB. Assume 12ns for each combinational CLB in the path including routing between CLBs. Since the XC3064 is one of the larger devices in the Xilinx XC3000 family, routing delays can be large. Therefore, assume an additional 3ns to give 15ns of combinational delay per non-registered CLB. (New, 1994, p. 8-36)

According to Xilinx's speed estimation method, the encoder design should be capable of a clock rate of about 15 MHz. With the addition of pipeline registers in the datapath, no datapath CLB is combinational. However, there is a combinational CLB at the output of both the LOADER and IN_ENBLE state machines that hurt performance. This problem should be addressed in future versions of this design. Thus, remembering that the control blocks and datapath blocks are clocked on opposite edges, the critical path has a delay of 15ns + 18ns or 33ns. Signals must traverse this delay in *one half* of the clock period; hence, the maximum clock rate of about 15 MHz.

2. APR Iterations

The programmable convolutional encoder was placed and routed three times. After each run, the back annotated LCA file was simulated in QuickSim, and the output waveforms were checked against the waveforms generated in the high level behavioral simulations. To estimate the maximum speed of the design, the clock period was repeatedly

58

decreased until the waveforms no longer matched. This value of the clock period is an estimate of the minimum clock period or maximum clock rate of the design. The maximum clock rate of the non-pipelined version is about 9.7 MHz.

The pipelined version of the encoder was placed and routed twice, with and without a user constraints file. Without the constraints file, the clock rate dropped to about 8.3 MHz showing that pipelining by itself does not necessarily improve performance. The routing delays must also be minimized by using a constraint file to tell the APR program which signals are timing critical. With the constraints file, APR produced an implementation with a maximum clock rate of about 11.1 MHz -- not much of an improvement.

Most likely, another APR run is needed which uses a very detailed constraint file that allows APR to do almost nothing for itself. The file will include locations of CLBs, which type of routing resource to use for the most critical signals, etc. Since I/O pin assignments affect placement and routing, all signals which were originally brought out to I/O pins for monitoring purposes should be eliminated

E. AUTOLOGIC

Ultimately the Mentor Graphics AutoLogic tool was not useful. The intent was to use it to generate schematics to partition into units that would fit in Xilinx CLBs. Even though the ECE Department does not have the Xilinx libraries for Autologic to use, the actual schematics generated were important only to gauge fan-in logic to CLBs, making the libraries unnecessary. The idea was to partition the design into blocks with five or fewer inputs and enter the modified schematics into the old Mentor Graphics suite that has the Xilinx library. Unfortunately, when a CAD tool automatically produces output, the designer is to some extent giving up control over the outcome of the design process. This became quite apparent when Autologic produced schematics that were very difficult to decipher. Tracing through some of the logic showed that the CAD tool had implemented lots of

redundant logic with long signal paths, defeating the purpose of one-hot encoding. Further, the schematics were very complicated and disorganized. At least when the designer draws his own schematic, he can organize it in his own accurate, intelligible way. Many menu permutations were tried, but there was almost no improvement. Manually translating VHDL source code into the proper schematic diagrams proved to be much more efficient and useful.

F. BACK ANNOTATION INTO VHDL CODE

One of the original reasons for using VHDL in this thesis was to back annotate timing information into the VHDL code. However, after using the Xilinx development system to examine the completed placement and routing inside the LCA, it became obvious that back annotation would not be worth the long, tedious process required. It is very inefficient and error prone to navigate through a routed LCA attempting to pick out the appropriate delays for back annotation. The only reason to do back annotation at all is for detailed documentation of a particular implementation of a design. To back annotate for VHDL simulation purposes is simply doubling the designer's work because the design can be back annotated into the QuicksimII simulator anyway. As Steve Carlson of Synopsis, Inc. points out (Harr, 1991, p. 149), back annotating timing information into the VHDL code ties the VHDL description to a specific technology, defeating one of the most important reasons for using VHDL: to have an accurate description or specification of system behavior that is completely independent of the target technologies in which the design can be implemented. The timing information (setup and hold times, clock rates, etc.) is different for each technology. It is up to the designer to guarantee that the design behaves according to the VHDL code within the particular timing constraints imposed by the specific target system and target technology.

60

VIII. CONCLUSION

Convolutional encoding is a Forward Error Correction (FEC) technique used in continuous one-way and real time communication links. It can provide substantial improvement in bit error rates so that small, low power, inexpensive transmitters can be used in such applications as satellites and hand-held communication devices. This thesis documents the development of a programmable convolutional encoder implemented in an Field Programmable Gate Array (FPGA) and capable of coding a digital data stream with any one of 39 convolutional codes. It has a simple microprocessor interface, a register file for storage of code parameters, a test circuit, and a maximum bit rate of about 15 Mbits/s.

The VHSIC Hardware Description Language (VHDL) is used to model abstract behavior and to define relationships between building blocks before hardware implementation in an XC3064 Logic Cell Array (LCA). An LCA is a type of FPGA made by Xilinx, Inc. Special design techniques like one-hot state assignment, pipelining, and exploitation of redundant states are employed to tailor the hardware to the LCA architecture. Because an FPGA is used for the hardware implementation, the design can be changed or expanded conveniently in the lab. In particularly flexible systems, several encoder designs can be stored in the system RAM, each one being downloaded into the FPGA under different circumstances.

More work can be done on this programmable convolutional encoder design. The bit rate can be increased substantially by decreasing the combinational delays in the paths of the "en" and "load" control signals. More sophisticated use of User Constraint files along with various options available in the Xilinx CAD programs should add to the performance improvement. Xilinx has published many application notes, all of which should be studied very closely to get the full benefit that the LCA architecture offers. One in particular,

61

entitled *Advanced Design Methodology* (Simpson, 1989), covers methods of independently placing and routing building blocks of the design. It is reasonable to expect the output bit rate to approach 30 Mbits/s after those more sophisticated techniques are employed. A register based FIFO memory can also be implemented on the LCA. This was mentioned in Chapter III, and will make the programmable encoder easier to integrate into other systems.

The top-down design paradigm proved to be very beneficial for this design. Several behavioral bugs were discovered and fixed at the beginning of the design cycle that would have been difficult to find in the later stages. More importantly, it forced a detailed definition of system partitioning and building block interaction before hardware became a factor in the design. Once all of the behaviors were defined, the hardware was easier to derive, and the effort was focused on optimizing the hardware design for the Xilinx LCA architecture. Actually implementing the design in a Xilinx LCA is easy if performance is not a significant concern; the APR program does all of the work and produces a mediocre result. If high performance is a concern, however, the designer must take control of the place and route process via User Constraint files, command line options available with APR, and advanced techniques outlined in the Xilinx application notes.

VHDL SOURCE CODE

A. SHIFTREG

```
LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;
USE MGC_PORTABLE.QSIM_RELATIONS.ALL;


-- This is the source code for the SHIFTREG block which is a 4-bit serial to parallel shift
-- register. It takes serial message bits as input at "serial_input" and provides them to the
-- DATAREG block in parallel. It is enabled by "en" from the IN_ENBLE block.


ENTITY shiftreg IS
  PORT (parallel_out1  : OUT BIT;
         parallel_out2  : OUT BIT;
         parallel_out3  : OUT BIT;
         parallel_out4  : OUT BIT;
         serial_input   : IN BIT;
         clk,reset,en   : IN BIT);
END shiftreg;


ARCHITECTURE arch1 OF shiftreg IS
  SIGNAL q : BIT_VECTOR(4 DOWNTO 1);
BEGIN
  PROCESS(clk, reset)
  BEGIN
    IF (reset = '0') THEN -- asynchronous reset.
      q <= "0000";
    ELSIF (clk'EVENT AND clk = '0') THEN
      IF (en = '1') THEN -- if "en" is active, shift contents
          q(4) <= q(3);    -- by 1 bit position and input new
          q(3) <= q(2);    -- message bit from "serial_input".
          q(2) <= q(1);
          q(1) <= serial_input;
      END IF;
    ELSE
    END IF;
  END PROCESS;

  parallel_out1 <= q(1); -- outputs of this block are
```

```vhdl
    parallel_out2 <= q(2);  -- the outputs of the flip-flops.
    parallel_out3 <= q(3);
    parallel_out4 <= q(4);

END arch1;
```

B. DATAREG

```vhdl
LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;
USE MGC_PORTABLE.QSIM_RELATIONS.ALL;


-- This is the source code for the DATAREG block which shifts k-tuples
-- through an 8-bit register. It provides the parallel output of the
-- register to the GENERATOR block.

ENTITY datareg4 IS
PORT (load : IN BIT;                  -- enables k-tuple loading into DATAREG
      k : IN BIT_VECTOR (2 DOWNTO 0);  -- defines message bits/k-tuple
      k_vect : OUT BIT_VECTOR (7 DOWNTO 0); -- parallel output of DATAREG
      reset : IN BIT;                 -- system reset
      clk : IN BIT;                   -- system clock
      in7, in6, in5, in4 : IN BIT);   -- parallel inputs from SHIFTREG
END datareg4;

ARCHITECTURE arch1 OF datareg4 IS
   SIGNAL d : BIT_VECTOR (7 DOWNTO 0);   -- D input to flip-flop
   SIGNAL q : BIT_VECTOR (7 DOWNTO 0);   -- Q output of flip-flop
BEGIN

-- Each D input gets the Q output of a different flip-flop depending on
-- the number of bits in a k-tuple.

   d(0)  <= q(1) WHEN (k = "001") ELSE  -- 1  bit/k-tuple, 1-bit shift
            q(2) WHEN (k = "010") ELSE  -- 2 bits/k-tuple, 2-bit shift
            q(3) WHEN (k = "011") ELSE  -- 3 bits/k-tuple, 3-bit shift
            q(4);                       -- 4 bits/k-tuple, 4-bit shift

   d(1)  <= q(2) WHEN (k = "001") ELSE
            q(3) WHEN (k = "010") ELSE
            q(4) WHEN (k = "011") ELSE
            q(5);

   d(2)  <= q(3) WHEN (k = "001") ELSE
            q(4) WHEN (k = "010") ELSE
            q(5) WHEN (k = "011") ELSE
            q(6);

   d(3)  <= q(4) WHEN (k = "001") ELSE
```

```vhdl
                  q(5) WHEN (k = "010") ELSE
                  q(6) WHEN (k = "011") ELSE
                  q(7);

   d(4)  <=  q(5) WHEN (k = "001") ELSE
                  q(6) WHEN (k = "010") ELSE
                  q(7) WHEN (k = "011") ELSE
                  in4;

   d(5)  <=  q(6) WHEN (k = "001") ELSE
                  q(7) WHEN (k = "010") ELSE
                  in5;  ·                          -- more than 2 bits/k-tuple

   d(6)  <=  q(7) WHEN (k = "001") ELSE
                  in6;                             -- more than 1 bit/k-tuple

   d(7)  <=  in7;                                  -- d(7) always gets in7

   k_vect <= q;                                    -- Q outputs of flip-flops form
                                                   -- parallel output of DATAREG.

PROCESS (clk, reset)
BEGIN
   IF (reset = '0') THEN                           -- asynchronous clear
      q <= "00000000";
   ELSIF (clk'EVENT AND clk = '0') THEN -- clock on falling edge only if
      IF (load = '0') THEN                         -- Q outputs get D inputs
         q<=d;                                     -- only if "load" input is low.
      END IF;
   ELSE
   END IF;
END PROCESS;
END arch1;
```

66

C. GENERATOR

```
LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;
USE MGC_PORTABLE.QSIM_RELATIONS.ALL;

-- The GENERATOR block provides 6 bits in parallel to the SEQUENCER block.
-- Each bit is the linear combination (bitwise AND followed by modulo-2 sum)
-- of a connection vector and the contents of the DATAREG block.

ENTITY generator IS
   PORT (g1, g2, g3, g4, g5, g6 : IN  BIT_VECTOR(7 DOWNTO 0); -- connection vectors
          k_vect : IN  BIT_VECTOR(7 DOWNTO 0);   -- parallel output of DATAREG
          mod2_sums : OUT BIT_VECTOR(6 DOWNTO 1)); -- output of GENERATOR
END generator;

ARCHITECTURE arch1 OF generator IS
BEGIN
   PROCESS(g1, g2, g3, g4, g5, g6, k_vect)
     VARIABLE sum : BIT_VECTOR (6 DOWNTO 1); -- each bit is a linear combination.
          -- "tempx"s hold result of bitwise AND.
     VARIABLE temp   : BIT_VECTOR (7 DOWNTO 0);
     VARIABLE temp2 : BIT_VECTOR (7 DOWNTO 0);
     VARIABLE temp3 : BIT_VECTOR (7 DOWNTO 0);
     VARIABLE temp4, temp5, temp6 : BIT_VECTOR (7 DOWNTO 0);
   BEGIN

     temp2 := g2 AND k_vect;
     temp3 := g3 AND k_vect;
     temp4 := g4 AND k_vect;
     temp5 := g5 AND k_vect;
     temp6 := g6 AND k_vect;

          -- sum of bitwise AND between k_vect and g1.
     sum(1) :=  temp1(7) XOR temp1(6) XOR temp1(5) XORtemp1(4) XOR
               temp1(3) XOR temp1(2) XOR temp1(1) XOR temp1(0);

          -- sum of bitwise AND between k_vect and g2.
     sum(2) :=  temp2(7) XOR temp2(6) XOR temp2(5) XOR temp2(4) XOR
               temp2(3) XOR temp2(2) XOR temp2(1) XOR temp2(0);

          -- sum of bitwise AND between k_vect and g3.
     sum(3) :=  temp3(7) XOR temp3(6) XOR temp3(5) XOR temp3(4) XOR
```

```vhdl
            temp3(3) XORtemp3(2) XOR temp3(1) XOR temp3(0);

       -- sum of bitwise AND between k_vect and g4.
 sum(4) :=  temp4(7) XOR temp4(6) XOR temp4(5) XOR temp4(4) XOR
            temp4(3) XOR temp4(2) XOR temp4(1) XOR temp4(0);

       -- sum of bitwise AND between k_vect and g5.
 sum(5) :=  temp5(7) XOR temp5(6) XOR temp5(5) XOR temp5(4) XOR
            temp5(3) XOR temp5(2) XOR temp5(1) XOR temp5(0);

       -- sum of bitwise AND between k_vect and g6.
 sum(6) :=  temp6(7) XOR temp6(6) XOR temp6(5) XOR temp6(4) XOR
            temp6(3) XOR temp6(2) XOR temp6(1) XOR temp6(0);

   mod2_sums  <= sum;        -- output is the six modulo-2 sums from above.

 END PROCESS;

END arch1;
```

D. IN_ENBLE

```
LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;
USE MGC_PORTABLE.QSIM_RELATIONS.ALL;


-- The IN_ENBLE block enables the SHIFTREG block long enough for SHIFTREG
-- to input k message bits. It inputs message bits while the SEQUENCER
-- block outputs code bits.


ENTITY in_enble IS
    PORT (k                 : IN BIT_VECTOR(2 DOWNTO 0); -- bits per k-tuple.
            clk,reset load  : IN BIT;                    -- system clock and reset.
            load            : IN BIT;                    -- "load" signal from LOADER block.
            en              : OUT BIT);                  -- enable to SHIFTREG.
    END in_enble;


ARCHITECTURE arch1 OF in_enble IS
    TYPE states IS (state_0, state_1, state_2, state_3, state_4);
    SIGNAL state : states;
BEGIN
    s:PROCESS(clk, reset)
     BEGIN
        IF (reset = '0') THEN        -- asynchronous reset.
            state <= state_0;
        ELSIF (clk'EVENT AND clk = '1') THEN -- if clk has changed, and it's
            CASE state IS                              -- now equal to '1', THEN...
                WHEN state_0 =>
                    IF (load = '0') THEN -- if "load" is active, go to state_1.
                        state <= state_1;
                    ELSE
                        state <= state_0;    -- stay in state_0 until "load" is inactive.
                    END IF;
                WHEN state_1 =>
                    IF ((k = "000") OR (k = "001")) THEN
                        state <= state_0;  -- if a k-tuple has 1 bit, go to state_0.
                    ELSE
                        state <= state_2;  -- k is more than 1.
                    END IF;
                WHEN state_2 =>
                    IF (k = "010") THEN  -- if a k-tuple has 2 bits, go to state_0.
                        state <= state_0;
                    ELSE
```

69

```vhdl
                state <= state_3;      -- k is more than 2.
            END IF;
          WHEN state_3 =>
            IF (k = "011") THEN  -- if a k-tuple has 3 bits, go to state_0.
              state <= state_0;
            ELSE
              state <= state_4;    -- k is more than 3.
            END IF;
          WHEN state_4 =>
              state <= state_0;    -- k is 4 or more.
        END CASE;
      ELSE
      END IF;
END PROCESS s;


PROCESS (state)
BEGIN
  IF (state = state_0) THEN
    en <= '0';            -- keep SHIFTREG disabled until "load" is active.
  ELSE
    en <= '1';            -- enable SHIFTREG, input serial message bits.
  END IF;
END PROCESS;

END arch1;
```

E. LOADER

```
LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;
USE MGC_PORTABLE.QSIM_RELATIONS.ALL;

ENTITY newload IS
  PORT (n        : IN BIT_VECTOR(2 DOWNTO 0);
        clk,reset : IN BIT;
        load      : OUT BIT);
END newload;
```

-- This is the source code for the LOADER block. LOADER outputs "load" which enables
-- DATAREG to input a k-tuple in parallel. It also synchronizes "en" which is the
-- output of the IN_ENBLE block that allows SHIFTREG to take serial data.

```
ARCHITECTURE arch1 OF newload IS
  TYPE states IS (state_1, state_2, state_3, state_4, state_5, state_6);
  SIGNAL state : states;
BEGIN
 s:PROCESS(clk, reset)
 BEGIN
   IF (reset = '0') THEN            -- asynchronous reset.
     state <= state_1;
   ELSIF (clk'EVENT AND clk = '1') THEN -- clock on rising edge.
     CASE state IS
       WHEN state_1 =>              -- go to state_2 regardless of n().
         state <= state_2;
       WHEN state_2 =>
         IF ((n = "000") OR (n = "001") OR (n = "010")) THEN
           state <= state_1;        -- go to state_1 if each n-tuple has 2 bits,
         ELSE                       -- defaults to n = 2 if n < 2.
           state <= state_3;        -- n has more than 2 bits.
         END IF;
       WHEN state_3 =>
         IF (n = "011") THEN        -- go to state_1 if each n-tuple has 3 bits.
           state <= state_1;
         ELSE
           state <= state_4;        -- n-tuple has more then 3 bits.
         END IF;
       WHEN state_4 =>
         IF (n = "100") THEN        -- go to state_1 if each n-tuple has 4 bits.
           state <= state_1;
```

71

```
          ELSE
            state <= state_5;          -- n-tuple has more than 4 bits.
          END IF;
        WHEN state_5 =>
          IF (n = "101") THEN          -- go to state_1 if each n-tuple has 5 bits.
            state <= state_1;
          ELSE
            state <= state_6;          -- n-tuple has 6 bits,
          END IF;                      -- defaults to 6 if n > 6.
        WHEN state_6 =>
          state <= state_1;
      END CASE;
    ELSE
    END IF;
  END PROCESS s;

  PROCESS (state)
  BEGIN
    IF (state = state_1) THEN
      load <= '0';          -- "load" output is active in state_1.
    ELSE
      load <= '1';          -- "load" is inactive in all other states.
    END IF;
  END PROCESS;

END arch1;
```

F. SEQUENCER

```
LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;
USE MGC_PORTABLE.QSIM_RELATIONS.ALL;

ENTITY sequencer3 IS
PORT (n            : IN BIT_VECTOR(2 DOWNTO 0);
      mod2_sums : IN BIT_VECTOR(6 DOWNTO 1);
      serial_output: OUT BIT;
      clk, reset   : IN BIT);
END sequencer3;

-- This is the source code for the SEQUENCER block. Depending on the
-- value of n, SEQUENCER selects the appropriate bits from "mod2_sums"
-- and outputs them through "serial_output" on the negative clock edge.

ARCHITECTURE arch1 OF sequencer3 IS
  TYPE states IS (state_0, state_1, state_2, state_3, state_4, state_5, state_6);
  SIGNAL state : states;
  SIGNAL serial : BIT := '0';
BEGIN
  s:PROCESS(clk, reset)
  BEGIN
    IF (reset = '0') THEN                    -- asynchronous reset.
      state <= state_0;
    ELSIF (clk'EVENT AND clk = '1') THEN -- state machine transitions on
      CASE state IS                          -- rising clock edge.
        WHEN state_0 =>                       -- go to state_1 regardless of the inputs.
          state <= state_1;
        WHEN state_1 =>                       -- go to state_2 regardless of the inputs.
          state <= state_2;
        WHEN state_2 =>
          IF (n="000" OR n="001" OR n="010") THEN -- if 2 bits/n-tuple, go to state_1,
            state <= state_1;                  -- n < 2 defaults to 2.
          ELSE
            state <= state_3;                  -- more than 2 bits/n-tuple.
          END IF;
        WHEN state_3 =>
          IF (n = "011") THEN
            state <= state_1;                  -- 3 bits/n-tuple.
          ELSE
            state <= state_4;                  -- more than 3 bits/n-tuple.
```

73

```vhdl
          END IF;
        WHEN state_4 =>
          IF (n = "100") THEN
            state <= state_1;      -- 4 bits/n-tuple.
          ELSE
            state <= state_5;      -- more than 4 bits/n-tuple.
          END IF;
        WHEN state_5 =>
          IF (n = "101") THEN
            state <= state_1;      -- 5 bits/n-tuple.
          ELSE
            state <= state_6;      -- more than 5 bits/n-tuple.
          END IF;
        WHEN state_6 =>
            state <= state_1;      -- 6 bits/n-tuple.(n>6 defaults to 6).
      END CASE;
    ELSE
    END IF;
  END PROCESS s;

-- mux structure that uses state flip-flops to select bits of "mod2_sums" for output.
  WITH state SELECT
  serial <= mod2_sums(1) WHEN state_1,
            mod2_sums(2) WHEN state_2,
            mod2_sums(3) WHEN state_3,
            mod2_sums(4) WHEN state_4,
            mod2_sums(5) WHEN state_5,
            mod2_sums(6) WHEN state_6,
            '0' WHEN state_0;

  PROCESS (clk, reset)
  BEGIN
    IF (reset = '0') THEN          -- asynchronous reset.
      serial_output <= '0';
    ELSIF (clk'EVENT AND clk = '0') THEN
      serial_output <= serial;     -- send code bits on falling edge of clock.
    END IF;
  END PROCESS;

END arch1;
```

G. HANDSHAK

```
LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;
USE MGC_PORTABLE.QSIM_RELATIONS.ALL;


-- HANDSHAK is a small state machine inside the REGFILE block.
-- It provides the handshaking mechanism for the data bus.


ENTITY handshak IS
    PORT (clk, reset: : IN BIT;      -- system clock and reset.
          AS         : IN BIT;      -- address strobe from microprocessor (active low).
          ASout      : OUT BIT;     -- enable signal for loading registers in DATAREG.
          DTACK      : OUT BIT);    -- response to AS back to microprocessor (active low).
END handshak;


ARCHITECTURE arch1 OF handshak IS
    CONSTANT state_1: BIT_VECTOR := "01";
    CONSTANT state_2: BIT_VECTOR := "11";
    CONSTANT state_3: BIT_VECTOR := "00";
    CONSTANT state_4: BIT_VECTOR := "10"; -- not used
    SIGNAL state : BIT_VECTOR (1 DOWNTO 0);
BEGIN
    a:PROCESS(clk, reset)
      BEGIN
        IF (reset = '0') THEN                 -- asynchronous reset.
          state <= state_1;
        ELSIF (clk'EVENT AND clk = '1') THEN  -- clock on rising edge.
          CASE state IS
            WHEN state_1 =>
              IF (AS = '0') THEN              -- if AS is active, go to state_2.
                state <= state_2;
              ELSE
                state <= state_1;            -- stay in state_1 until AS is active.
              END IF;
            WHEN state_2 =>
              state <= state_3;              -- go to state_3 regardless of AS.
            WHEN state_4 =>                   -- state_4 is not used.
            WHEN state_3 =>
              IF (AS = '1') THEN             -- when AS becomes inactive, go to state_1.
                state <= state_1;
              ELSE
                state <= state_3;            -- stay in state_3 until AS is inactive.
```

75

```
            END IF;
        END CASE;
      ELSE
      END IF;
END PROCESS a;

b:PROCESS (state)
  BEGIN
    IF (state = state_1) THEN        -- state_1 is the idle state, so
      ASout <= '0';                  -- no registers are enabled, and
      DTACK <= '1';                  -- DTACK is inactive.
    ELSIF (state = state_2) THEN
      ASout <= '1';                  -- enable a register for 1 cycle.
      DTACK <= '1';
    ELSIF (state = state_3) THEN
      ASout <= '0';                  -- disable the register.
      DTACK <= '0';                  -- tell microprocessor that data has
    ELSE                             -- been accepted.
    END IF;
END PROCESS b;

END arch1;
```

H. STIMULUS

```
LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;
USE MGC_PORTABLE.QSIM_RELATIONS.ALL;


-- The STIMULUS block provides the test message "info_vector" to the
-- encoder. The connection vectors and k and n are changed manually
-- to simulate different codes and constraint lengths. Of course, this
-- file should be recompiled and updated in the testbench TEST7.
-- This block takes "en" as an input from the IN_ENBLE block and indexes
-- through the bits of the test message only when "en" is high.

ENTITY stimulus IS
    PORT (k: OUT BIT_VECTOR(2 DOWNTO 0);   -- bits in a k-tuple.
          n: OUT BIT_VECTOR(2 DOWNTO 0);   -- bits in an n-tuple.
          clk, reset: OUT BIT;                     -- asynchronous reset, system clock.
          serial_input: OUT BIT;                   -- test message output.
          -- connection vectors.
          G1, G2, G3, G4, G5, G6: OUT BIT_VECTOR(7 DOWNTO 0);
          en: IN BIT);                             -- "en" from IN_ENBLE block.
END stimulus;

ARCHITECTURE arch1 OF stimulus IS

    SIGNAL clock,rst,flag : BIT := '0';
    SIGNAL k_index : INTEGER := 17; -- index for traversing bit-by-bit
                                    -- through info_vector.
    CONSTANT info_vector : BIT_VECTOR (17 DOWNTO 1) := "10011101010000000";
BEGIN
rst  <= '1' AFTER 0ns,     -- resets chosen arbitrarily to make sure the encoder functions
        '0' AFTER 175ns, -- properly after an asynchronous reset.
        '1' AFTER 375ns,
        '0' AFTER 11375 ns,
        '1' AFTER 11575 ns;


 reset <= rst;

  k <= "001";
  n <= "010";


  G1 <= "10001101";
```

```vhdl
  G2 <= "10101001";
  G3 <= "10100010";
  G4 <= "10011110";
  G5 <= "10010011";
  G6 <= "10111101";

c:PROCESS
BEGIN                -- Generate clock (arbitrarily chosen at 10 MHz).
  clock <= '0';
  WAIT FOR 50ns;
  clock <= '1';
  WAIT FOR 50ns;
END PROCESS c;

clk <= clock;

PROCESS
BEGIN
  WAIT ON clock, rst;
    IF (rst = '0') THEN            -- If reset is active,
      k_index <= 17;              -- set index to first bit,
      serial_input <= '0';        -- and zero the output.
      WAIT FOR 5 ns;              -- Update signals.
    ELSIF (clock = '1') THEN      -- Otherwise, if rising clock edge,
      IF (en ='1') THEN           -- and "en" from IN_ENBLE is active, then
        IF (k_index = 1) THEN     -- check the index. If it is already at 1,
          k_index <= 1;           -- then keep it there and
          flag <= '1';            -- set a flag.
        ELSE                      -- If the index is not at 1,
          k_index <= k_index-1;   -- decrement it (go to next bit).
        END IF;
      ELSE
        IF (k_index = 1 AND flag = '1') THEN -- If "en" not active, and the final bit was
          k_index <= 17;          -- transmitted last time "en" was active (flag=1), then set
          flag <= '0';            -- index to first bit and reset the flag.
        END IF;
      END IF;
      WAIT FOR 5 ns;              -- Update signals.
      serial_input <= info_vector(k_index);-- Send current bit of info_vector to output.
      WAIT FOR 5 ns;              -- Update signals.
    END IF;
END PROCESS;
END arch1;
```

APPENDIX B

BEHAVIORAL TESTBENCH AND TIMING DIAGRAMS

A. BEHAVIORAL TESTBENCH

B. TIMING DIAGRAMS

1. **Figure B.1.**

    Timing diagram for a rate 1/6 code.

2. **Figure B.2.**

    Timing diagram for a rate 1/2 code.

3. **Figure B.3.**

    Timing diagram for a rate 3/5 code.

4. **Figure B.4.**

    Timing diagram for a rate 2/3 code.

BEHAVIORAL TESTBENCH

Figure B.1. Timing diagram for a rate 1/6 code

81
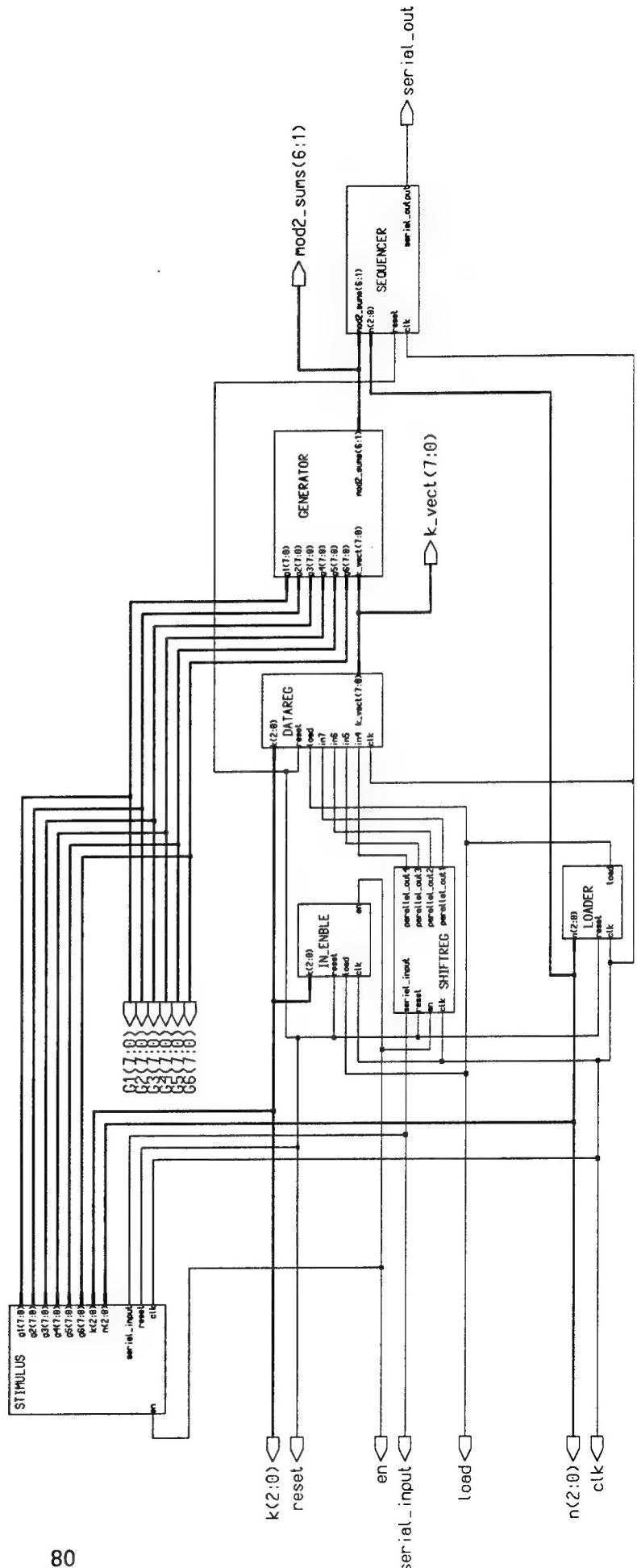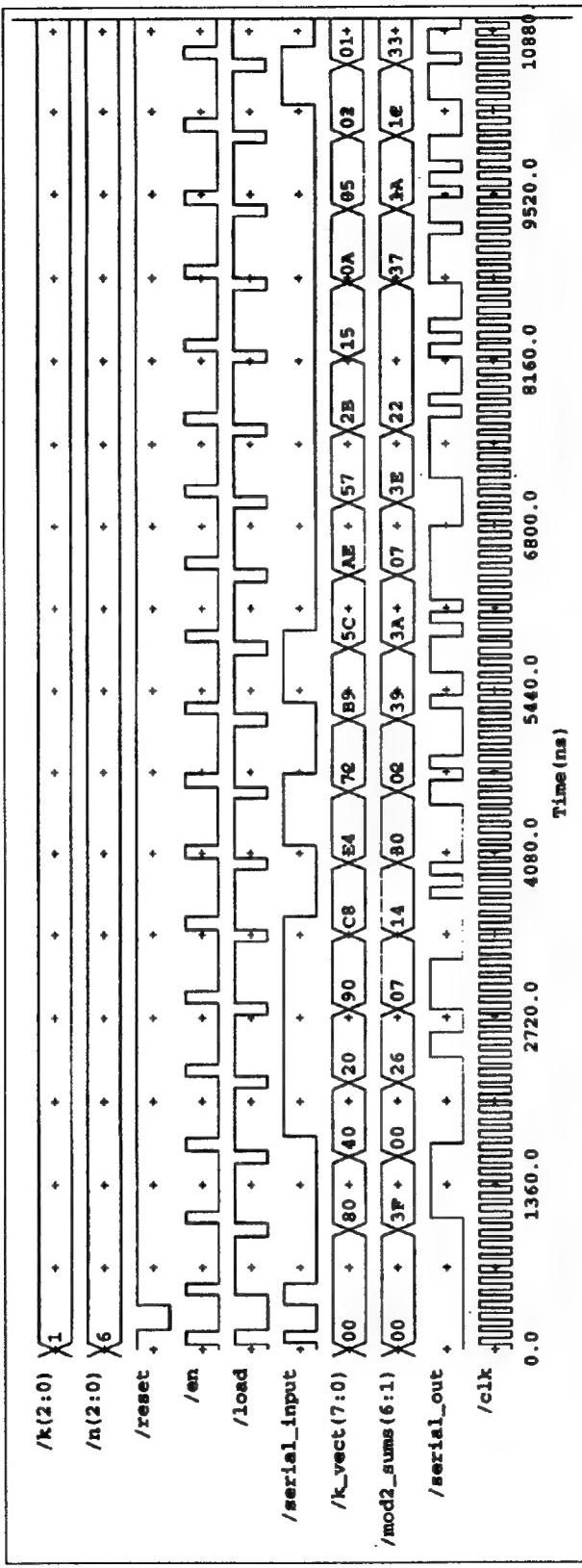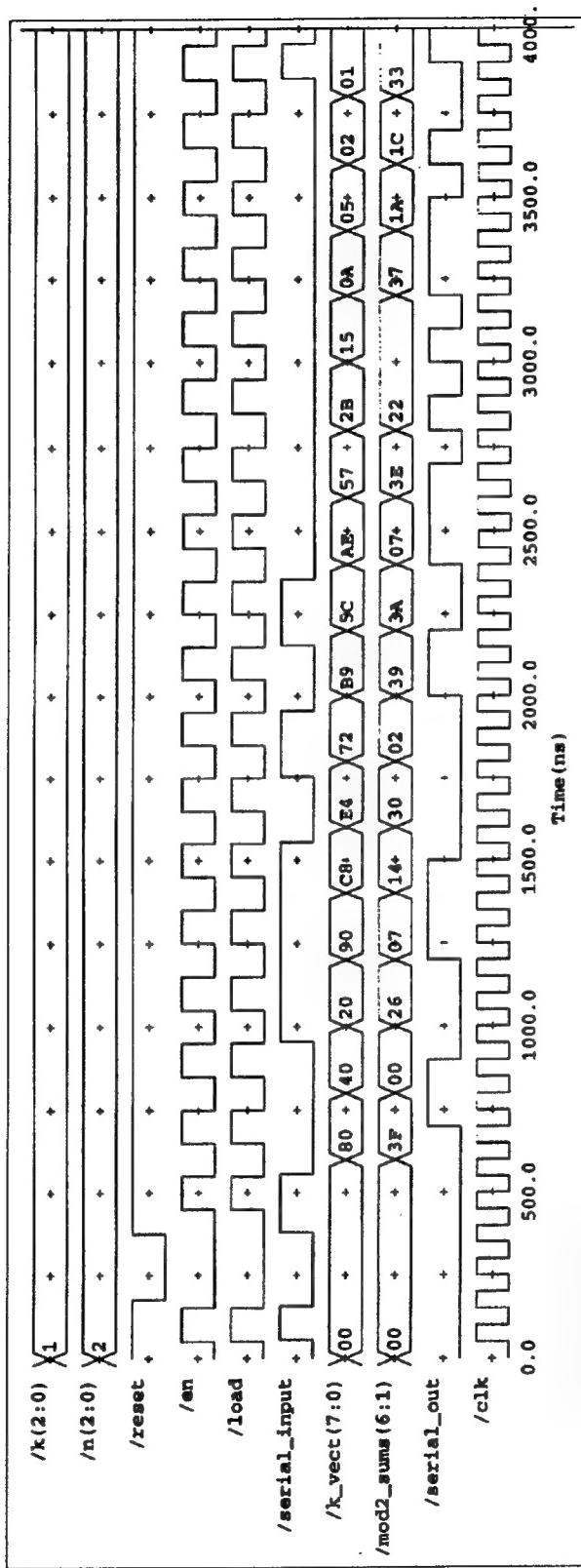
Figure B.2. Timing diagram for a rate 1/2 code

Figure B.3. Timing diagram for a rate 3/5 code

83

Figure B.4. Timing diagram for a rate 2/3 code

84
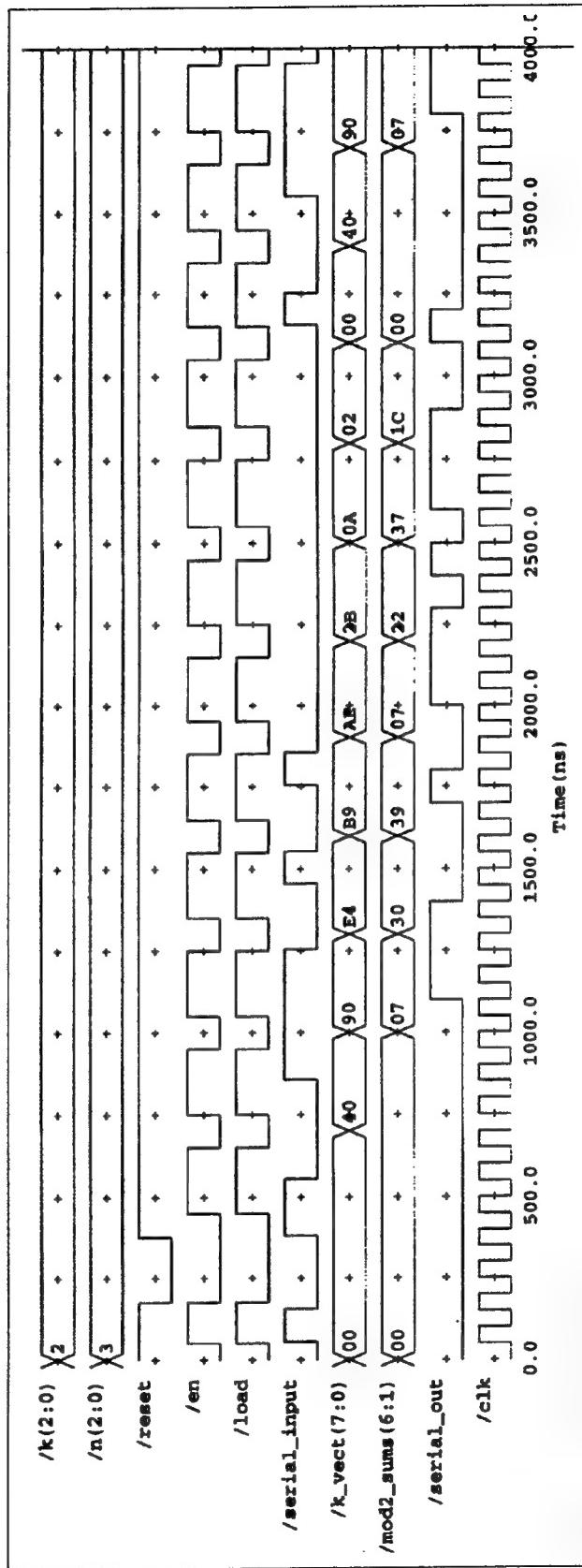
APPENDIX C

STATE DIAGRAMS FOR LCA
IMPLEMENTATIONS OF STATE MACHINES

A. IN_ENBLE

Figure C.1. State diagram for IN_ENBLE (one-hot, redundant state s0).

B. LOADER

Figure C.2. State diagram for LOADER (one-hot, redundant state s1).

C. SEQUENCER

Figure C.3. State diagram for SEQUENCER (one-hot, redundant state s1).

Figure C.1. State diagram for IN_ENBLE (one-hot, redundant state s0).

Figure C.2. State diagram for LOADER (one-hot, redundant state s1).

86
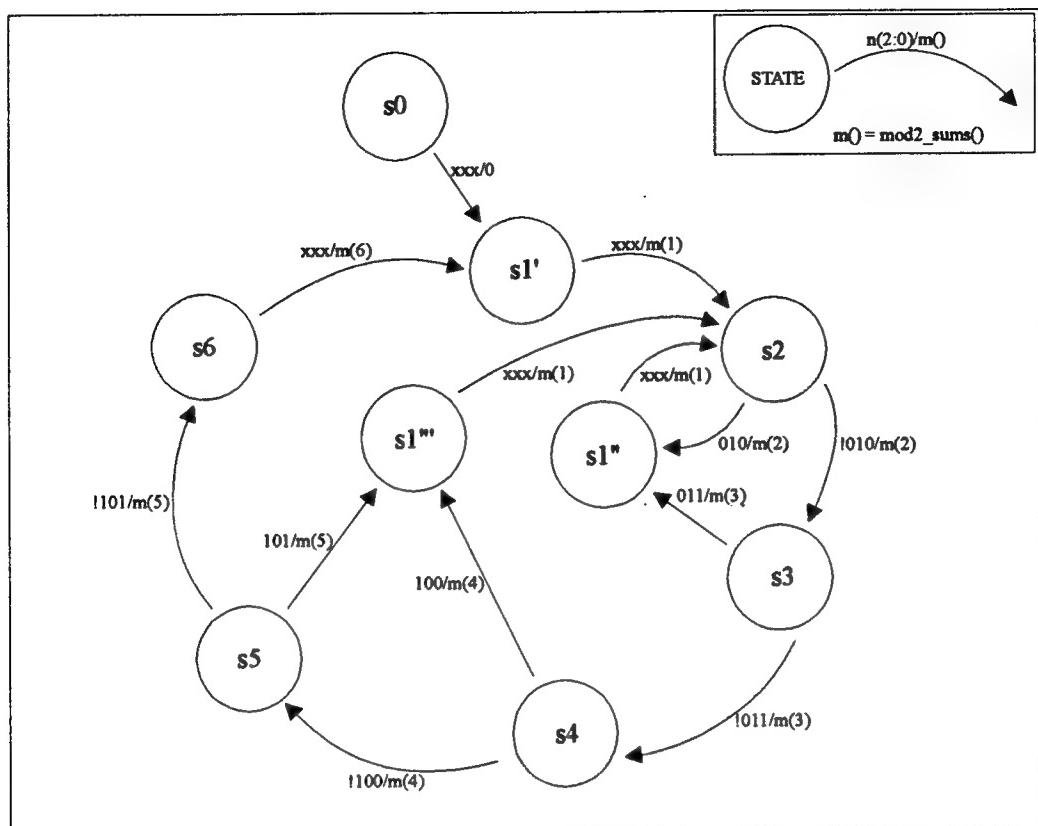
Figure C.3. State diagram for SEQUENCER (one-hot, redundant state s1).

APPENDIX D

SCHEMATIC DIAGRAMS FOR NON-PIPELINED PROGRAMMABLE CONVOLUTIONAL ENCODER

A. PROGRAMMABLE CONVOLUTIONAL ENCODER

B. ENCODER

C. SHIFTREG

D. DATAREG

E. GENERATOR

F. SEQUENCER

G. LOADER

H. IN_ENBLE

I. REGFILE

J. HANDSHAK

K. MUX

L. TEST

PROGRAMMABLE CONVOLUTIONAL ENCODER

ENCODER

mod2_sums (6:1)
kvect (7:0)
serial_out
en
load
dtack

TEST
test_vector (7:0)   test_input
CE
clk
reset

REGFILE
data(7:0)
select(2:0)
RS
clk
reset

MUX2
serial_input
A
B
S

test_vector (7:0)
clk
reset
data(2:0)
select(2:0)
RS

serial_input

ENCODER

SHIFTREG

DATAREG

clk

load

k (2:0)

in4

in5

in6

in7

reset

k_vect (7:0)

Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7

GENERATOR

SEQUENCER

clk

reset

n (2:0)

mod2_suns (6:1)

94

LOADER

IN_ENBLE

REGFILE

HANDSHAK

MUX

AND201 AND OR2 OR serial_input

A S B

TEST

serial_input

test_vector (7:0)

CE

clk

reset

M8_1E

D0
D1
D2
D3
D4
D5
D6
D7
S0
S1
S2
EN

Q

INV1

C16BCRD

0
1
2
3

CE
C

TC

RD
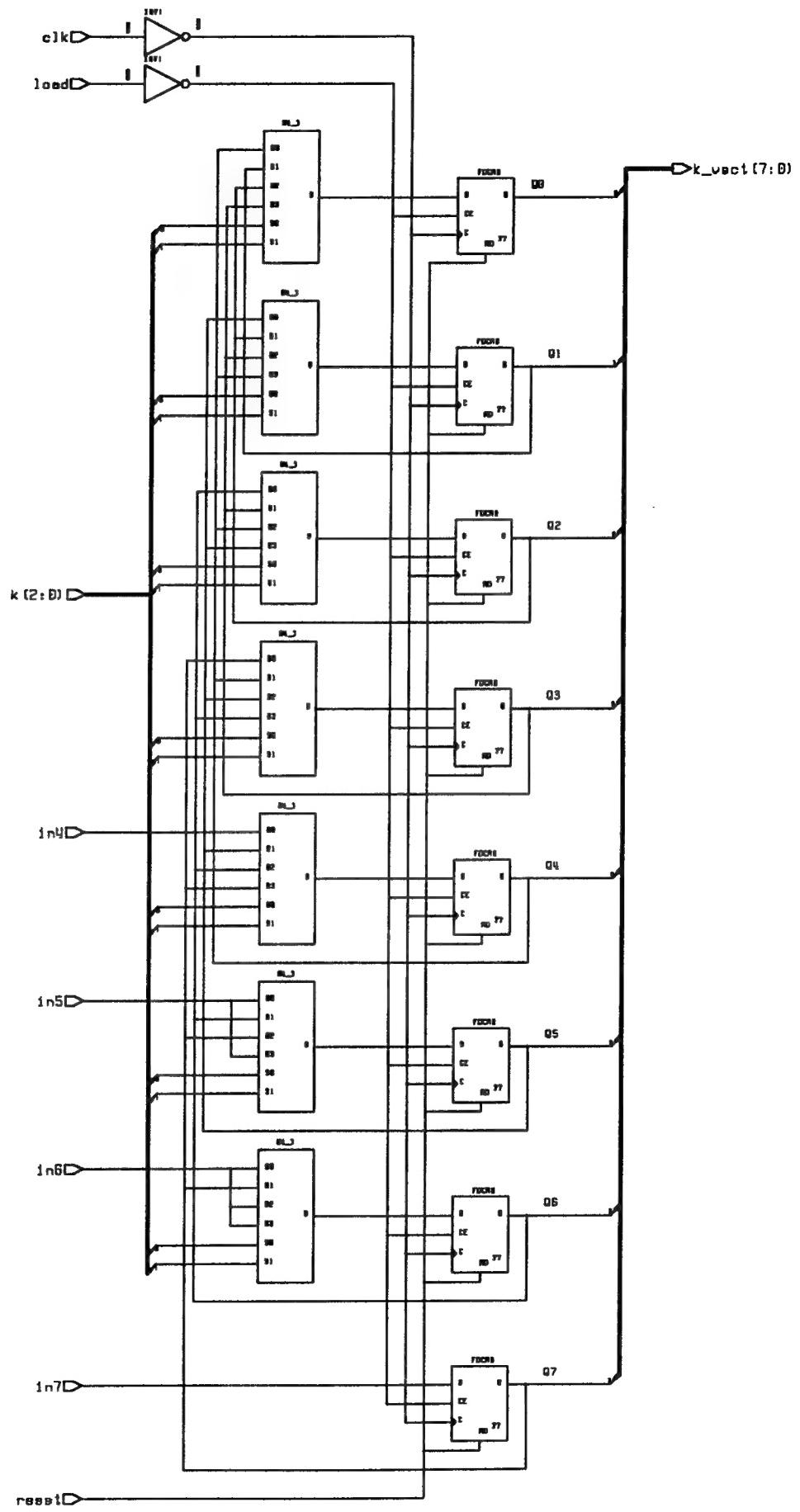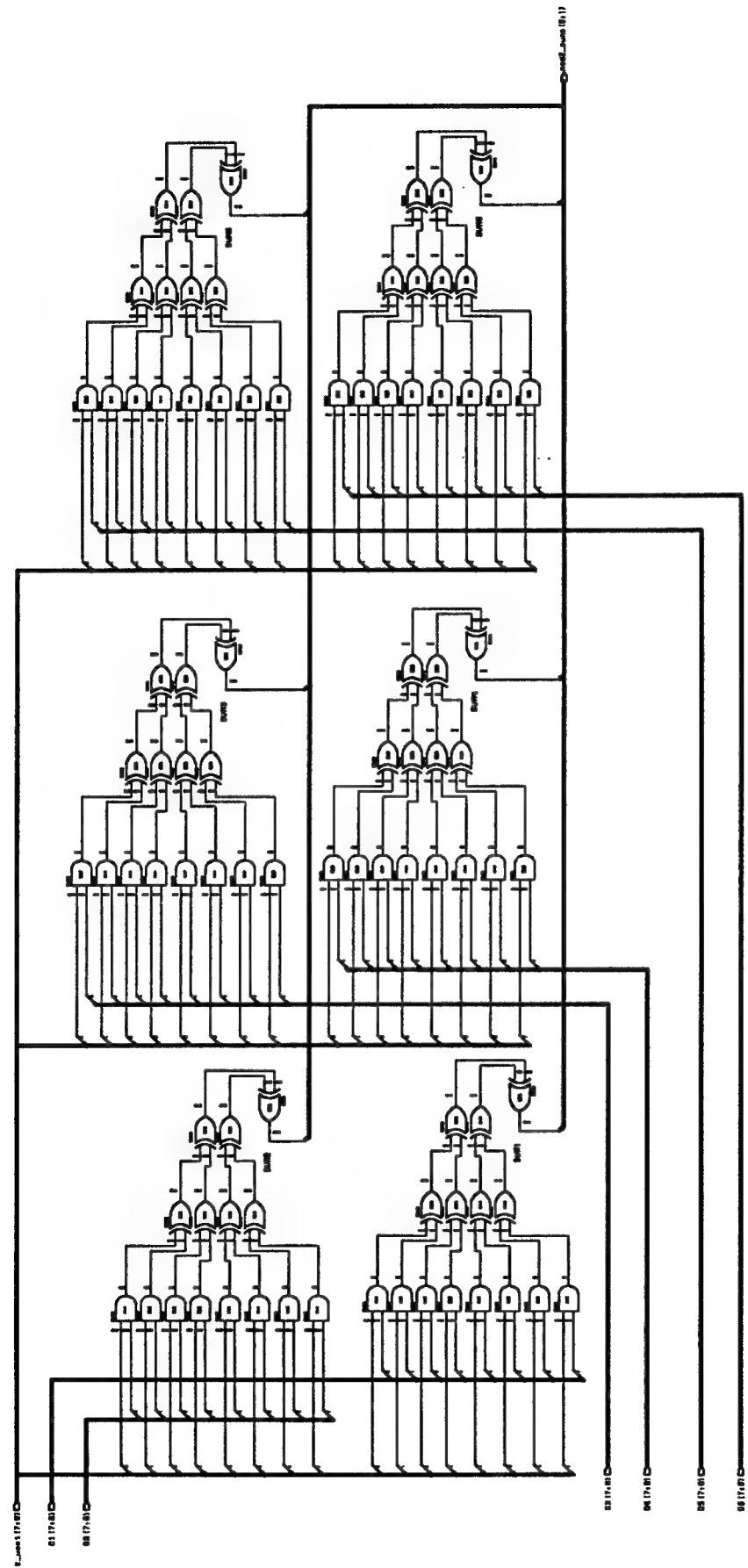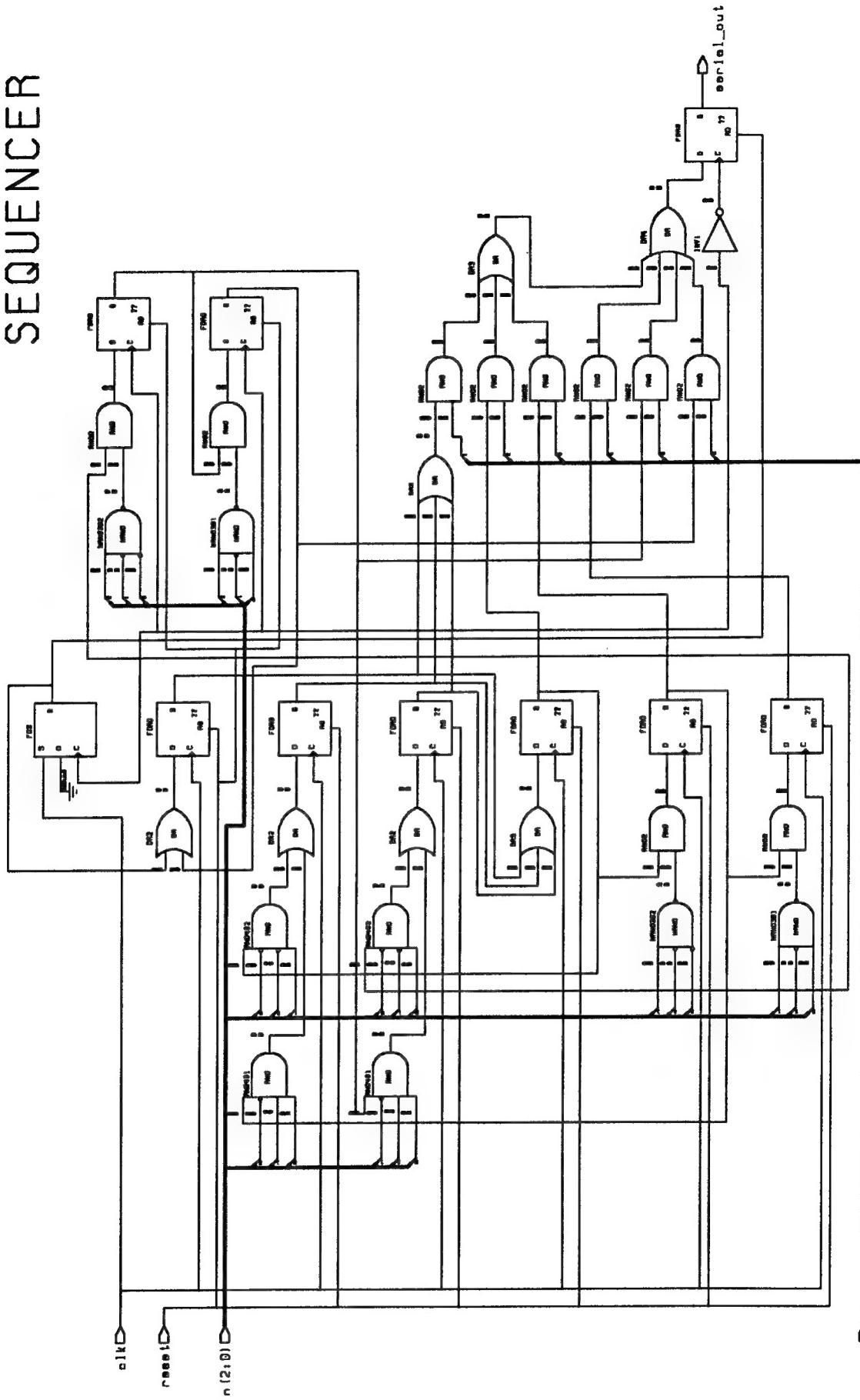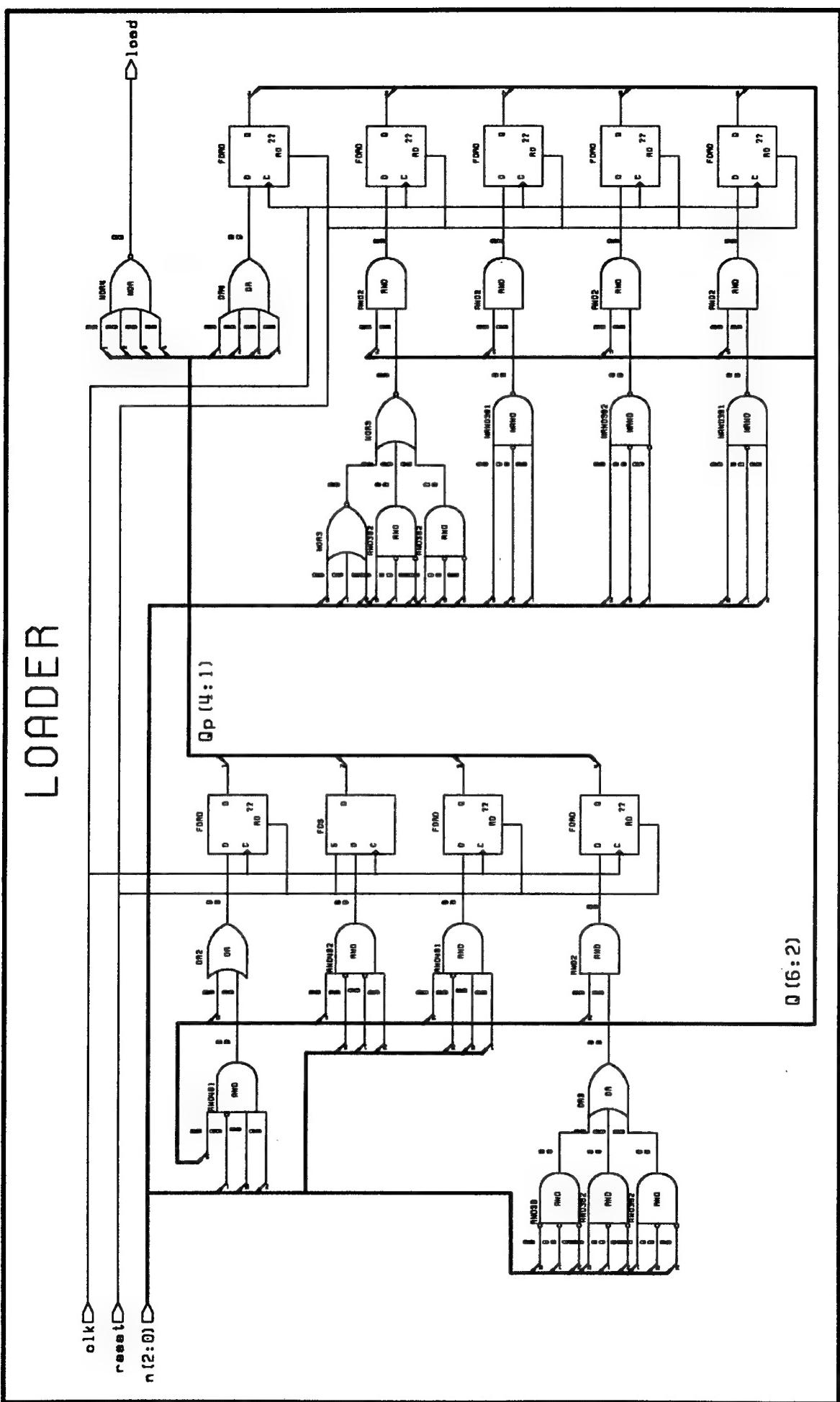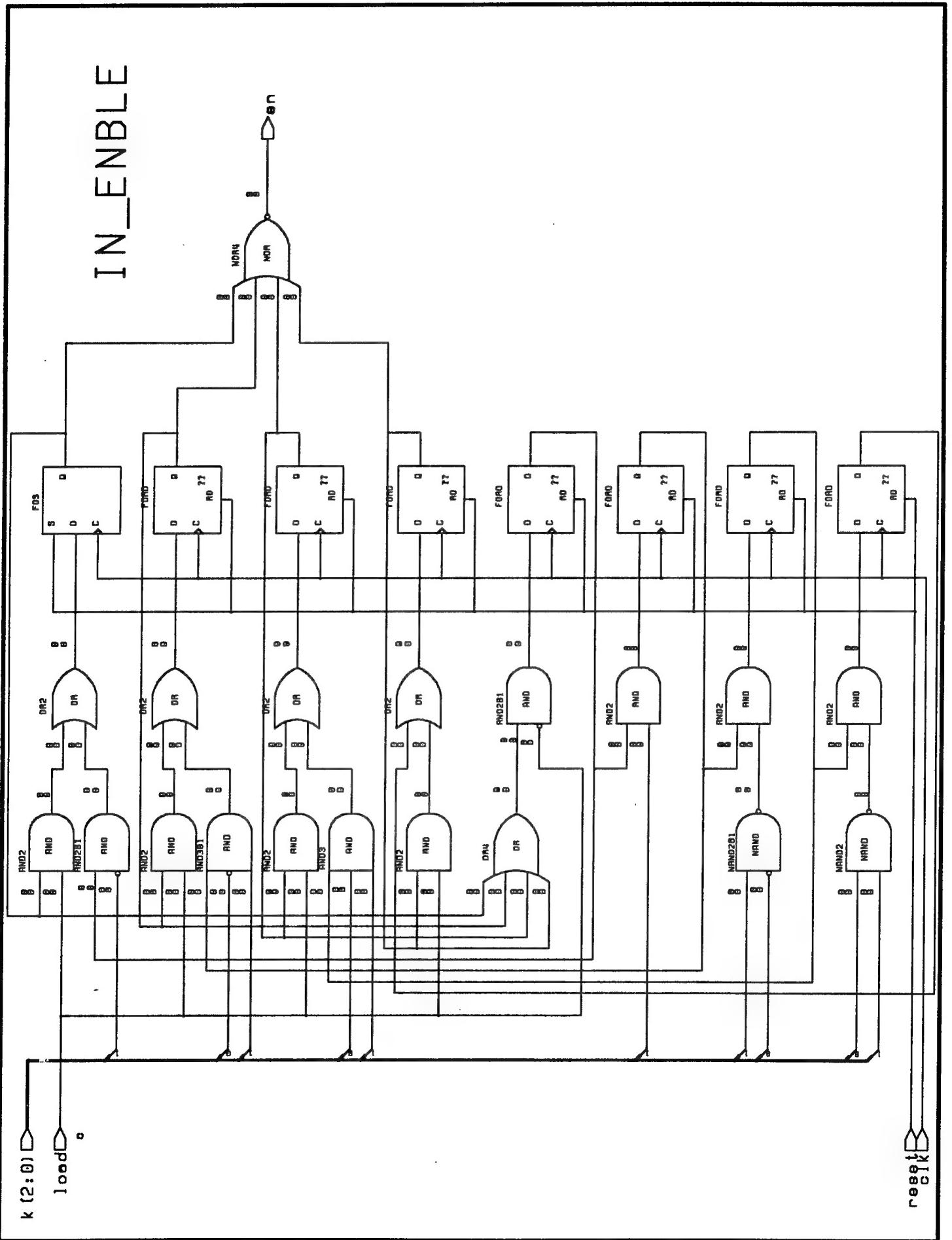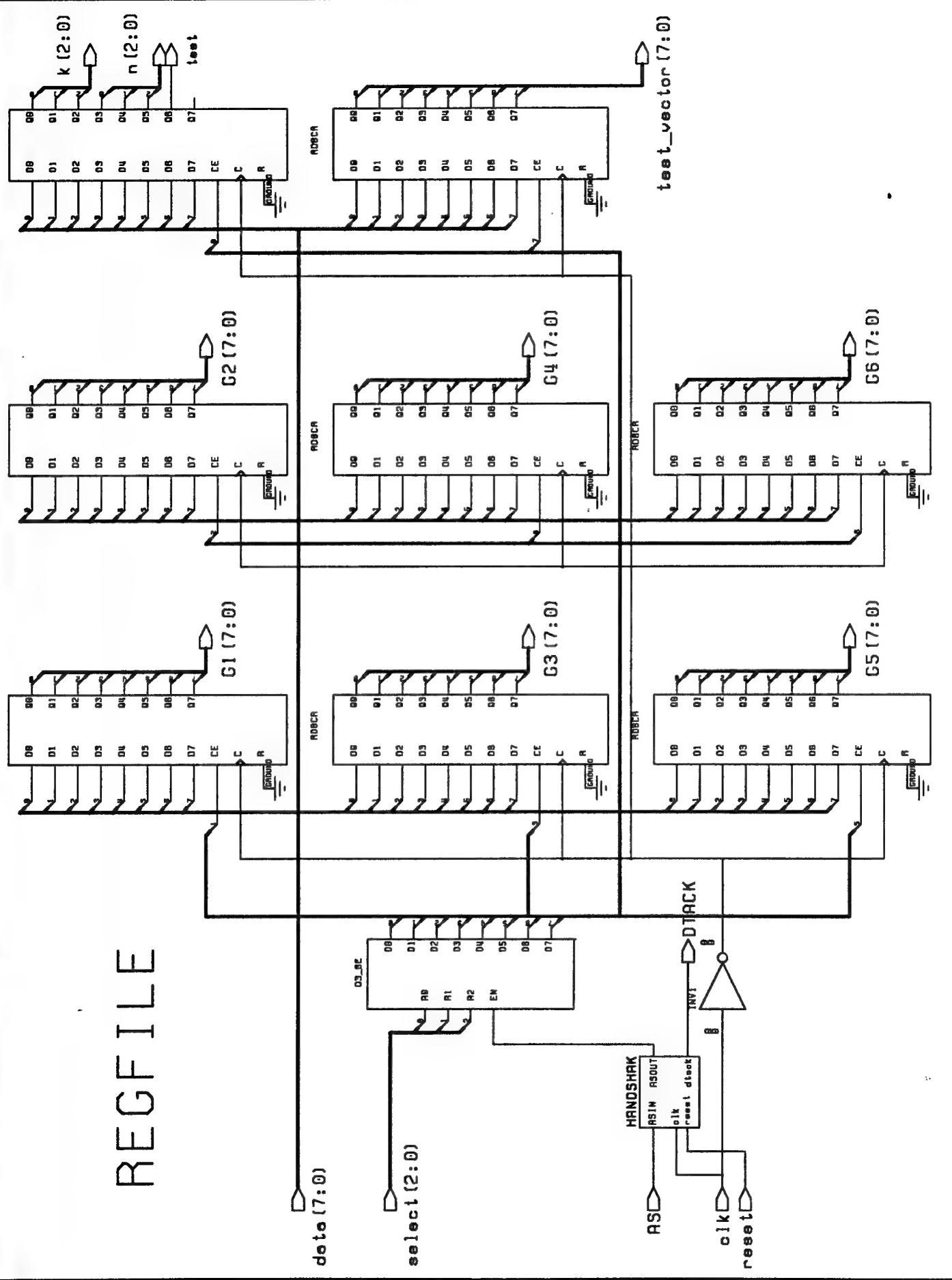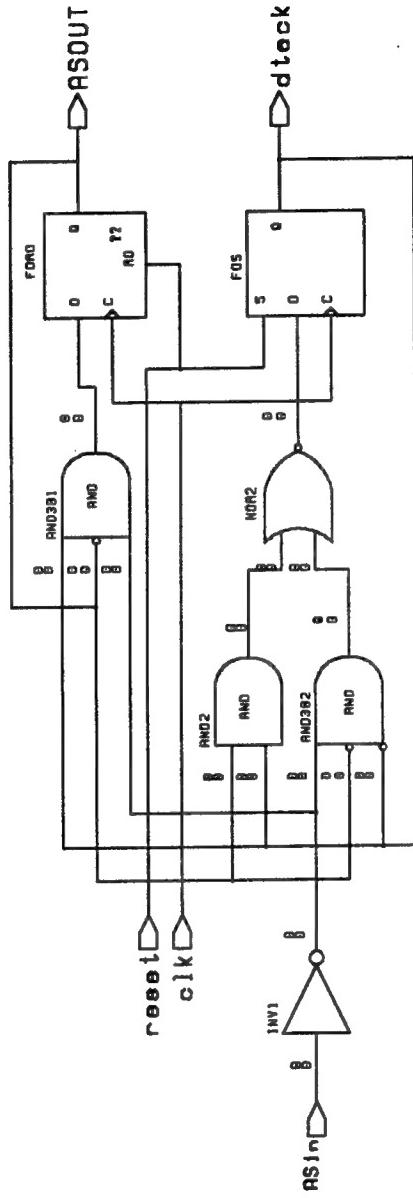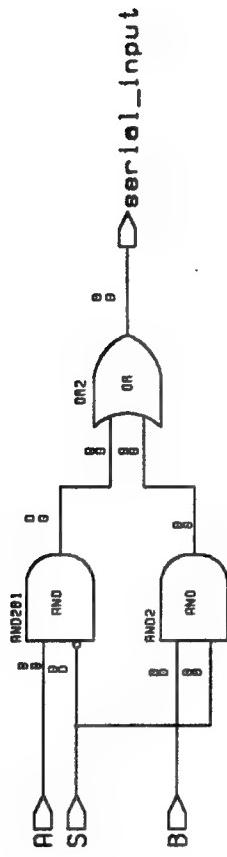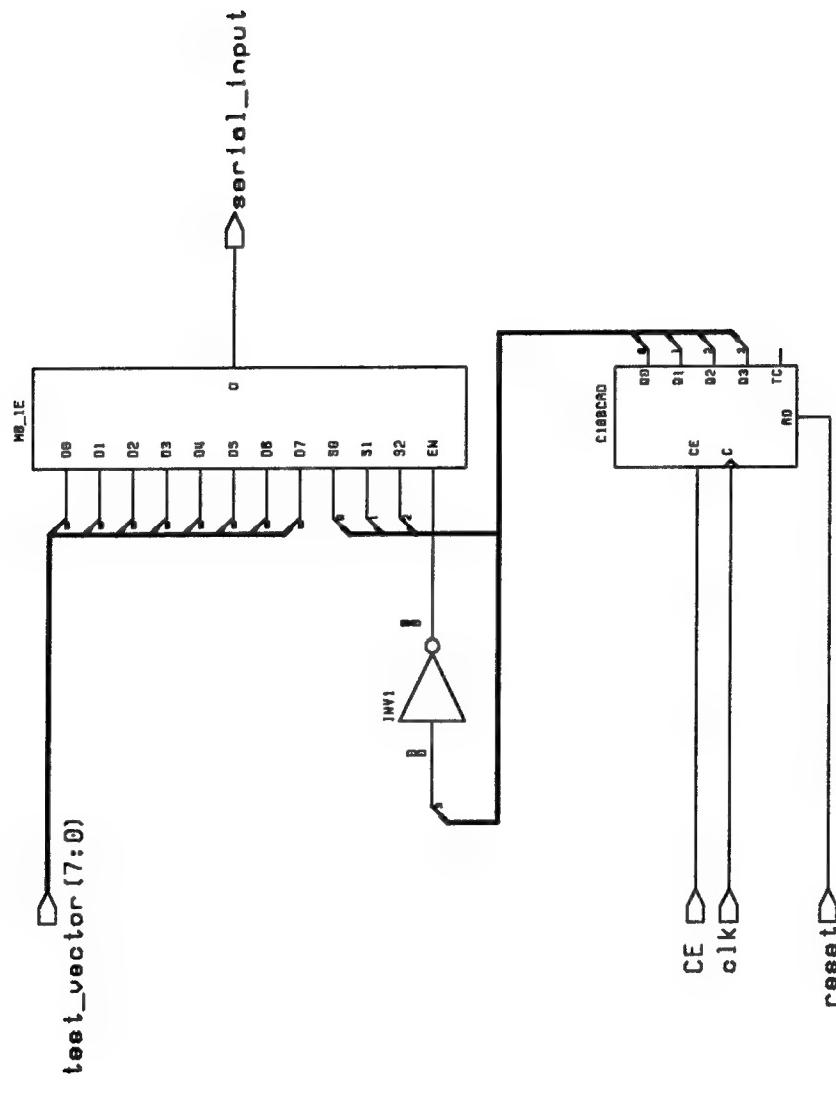
APPENDIX E

SCHEMATIC DIAGRAMS FOR PIPELINED BLOCKS
OF PROGRAMMABLE CONVOLUTIONAL ENCODER

This Appendix contains schematic diagrams only for the blocks to which pipeline registers were added. No other block was changed. All of the other schematics are in Appendix D.

A. ENCODER

B. GENERATORDFF

C. PIPESEQUENCER

ENCODER

GENERATORDFF

PIPESEQUENCER

clk

reset

n(2:0)

serial_out

mod2_sum(5:1)
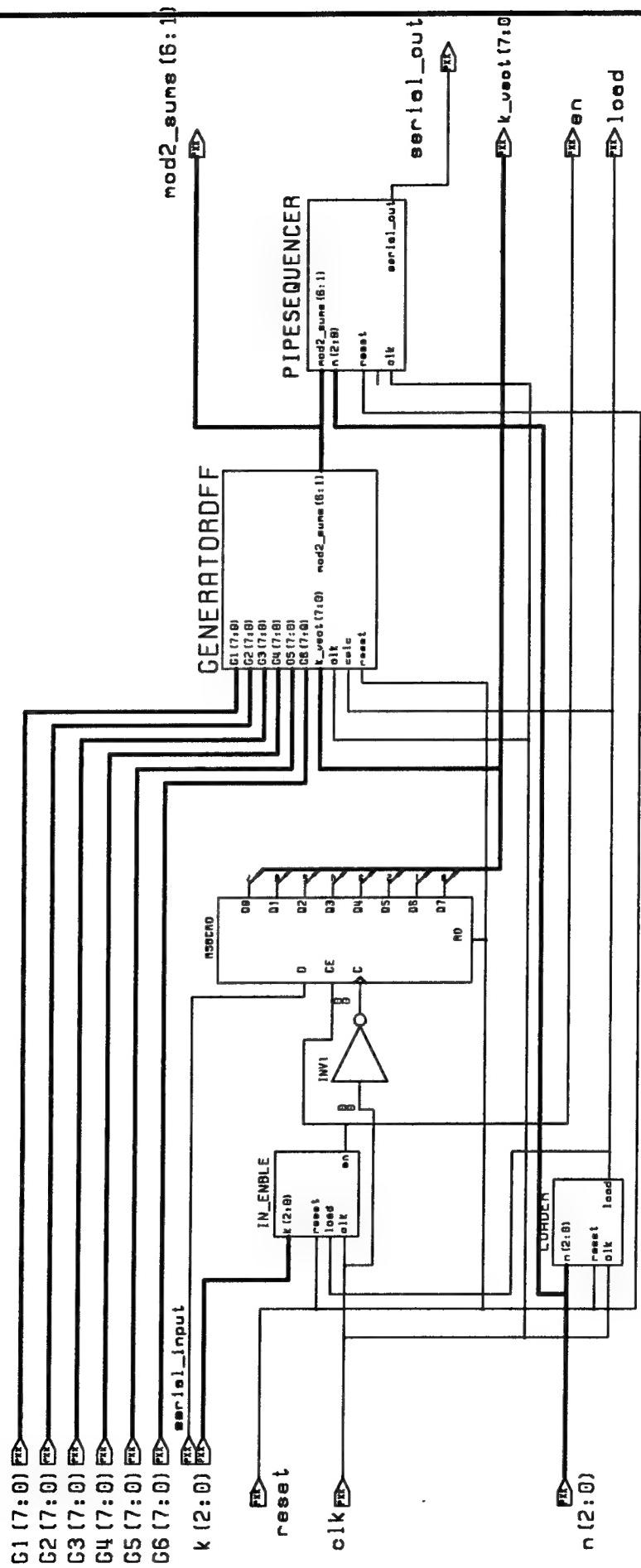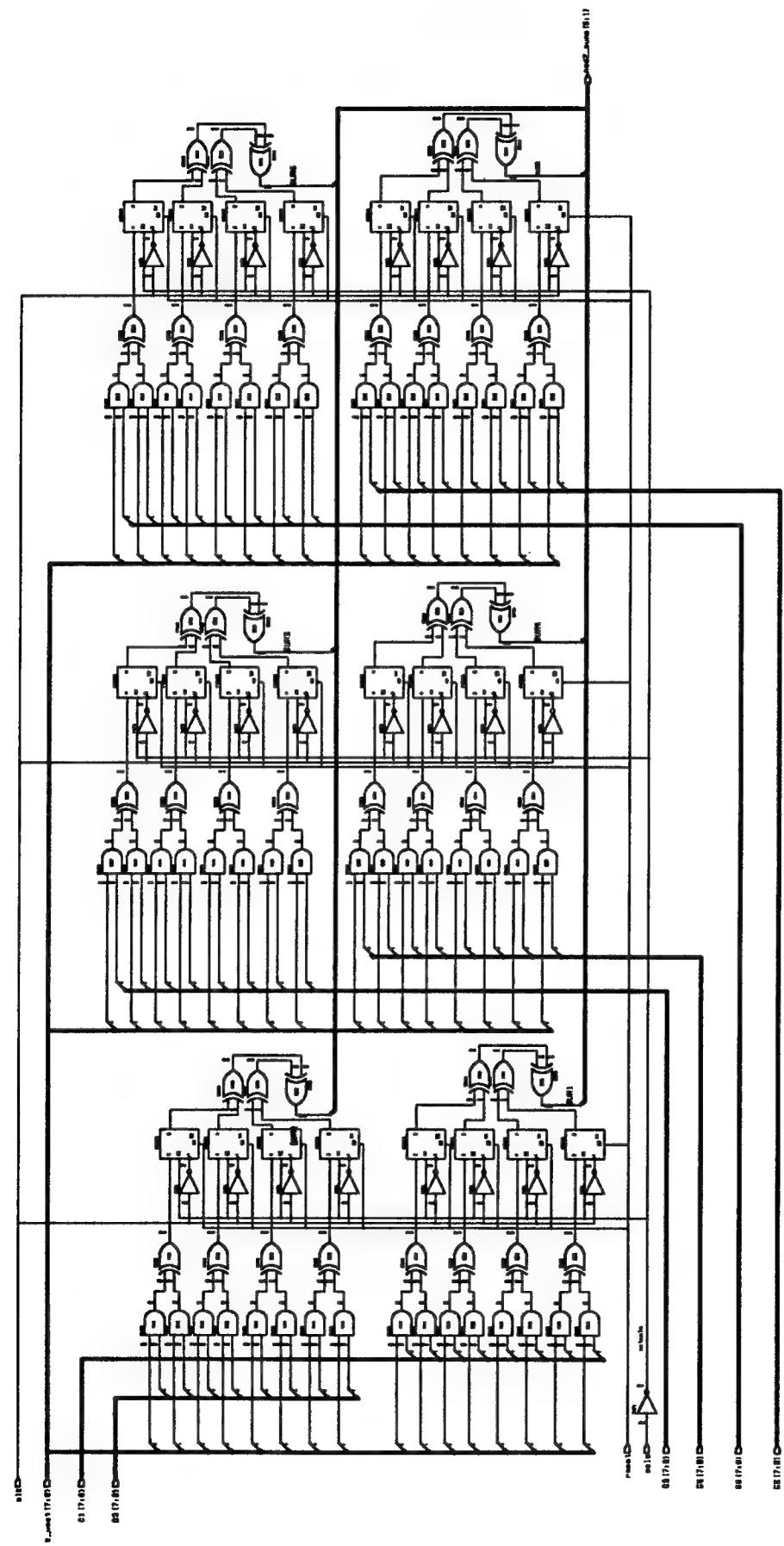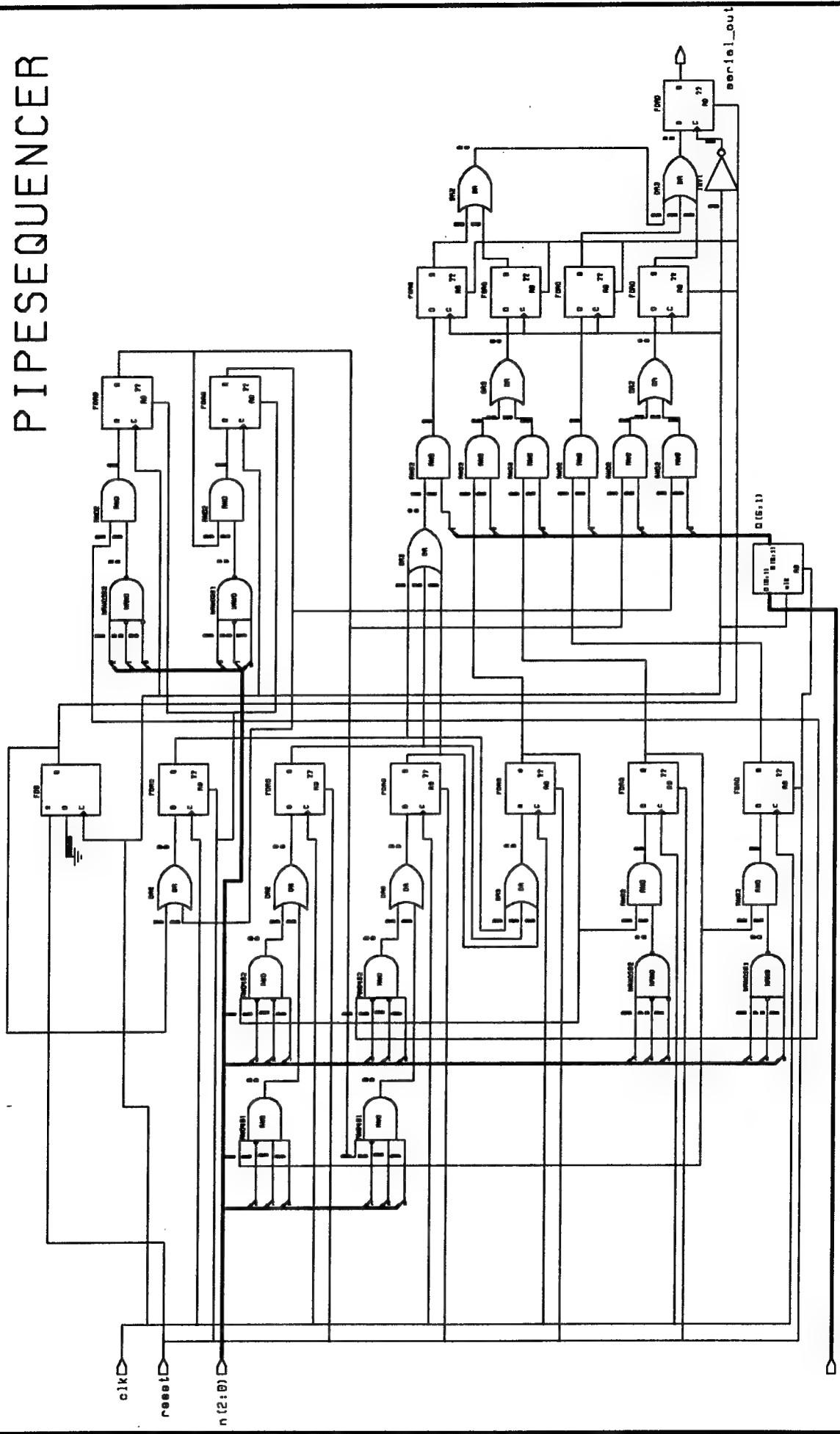
APPENDIX F

HARDWARE TESTBENCHES AND OUTPUT WAVEFORM

A. SCHEMATIC DIAGRAMS

1. Testbench

Schematic diagram of testbench for the encoder before placement and routing.

2. Testbenchb

Schematic diagram of testbench for the encoder after placement and routing.

B. OUTPUT WAVEFORM

1. Figure F.1.

Output waveform for back annotated encoder design.

TESTBENCH
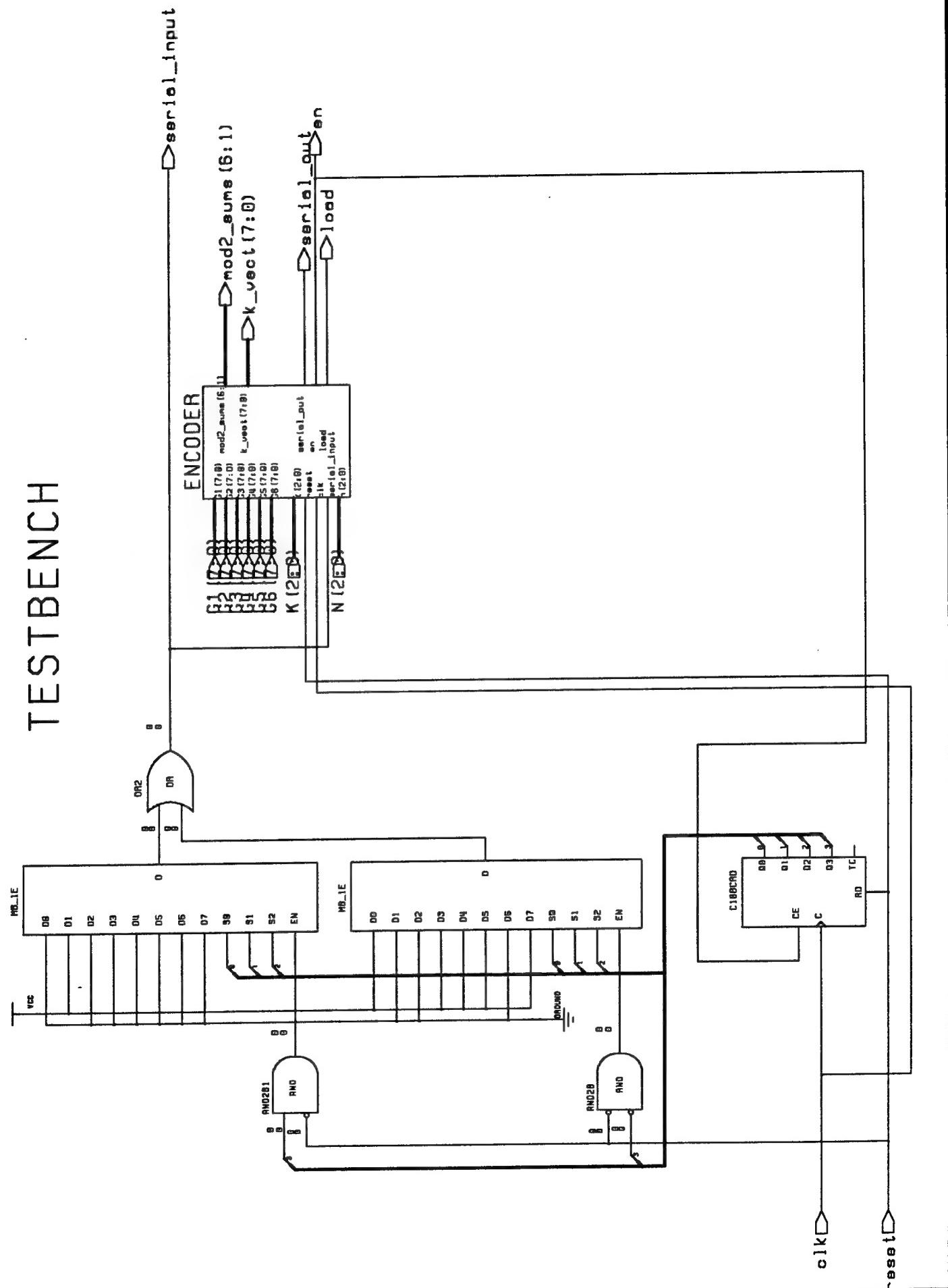
TESTBENCHB
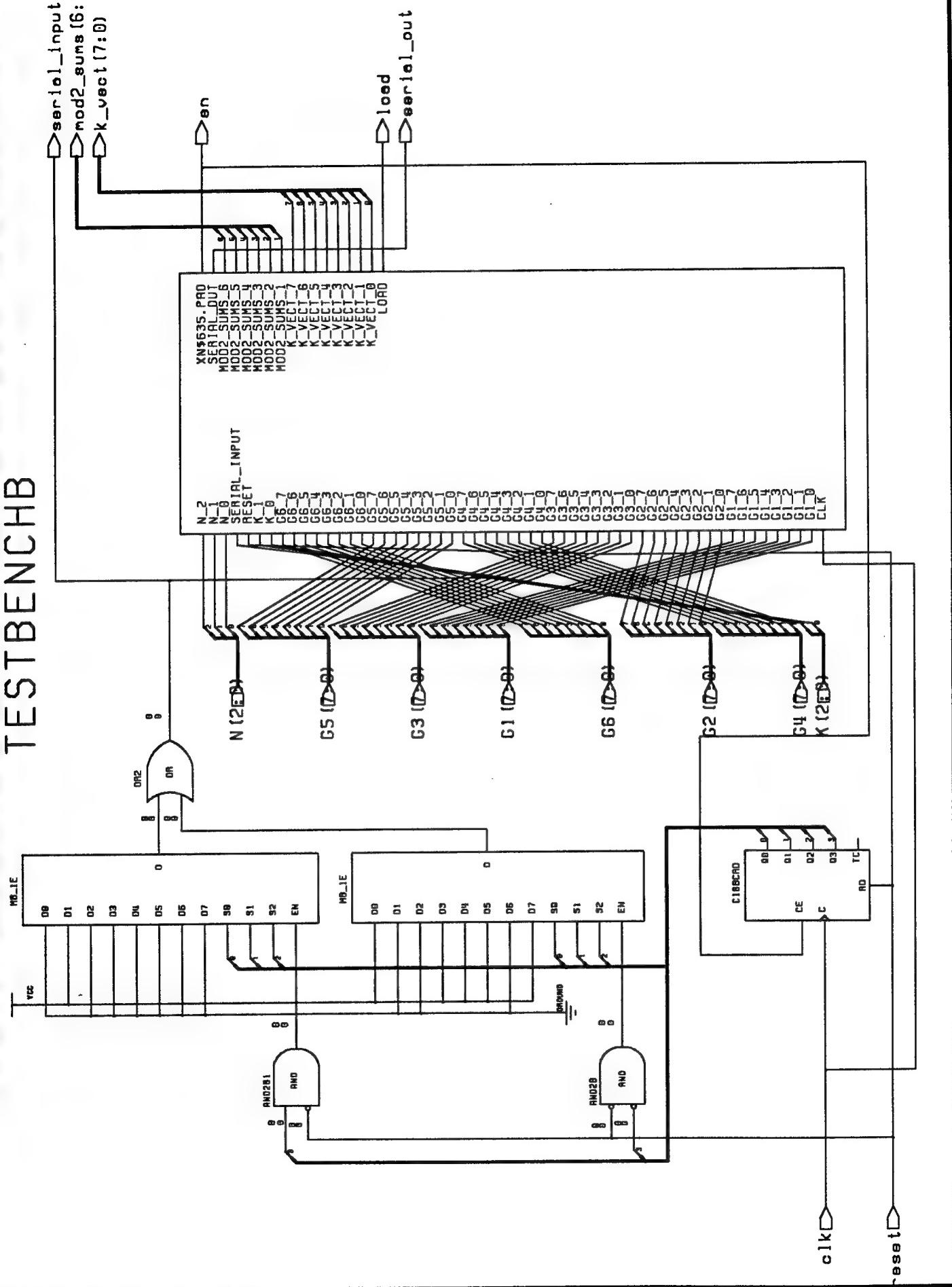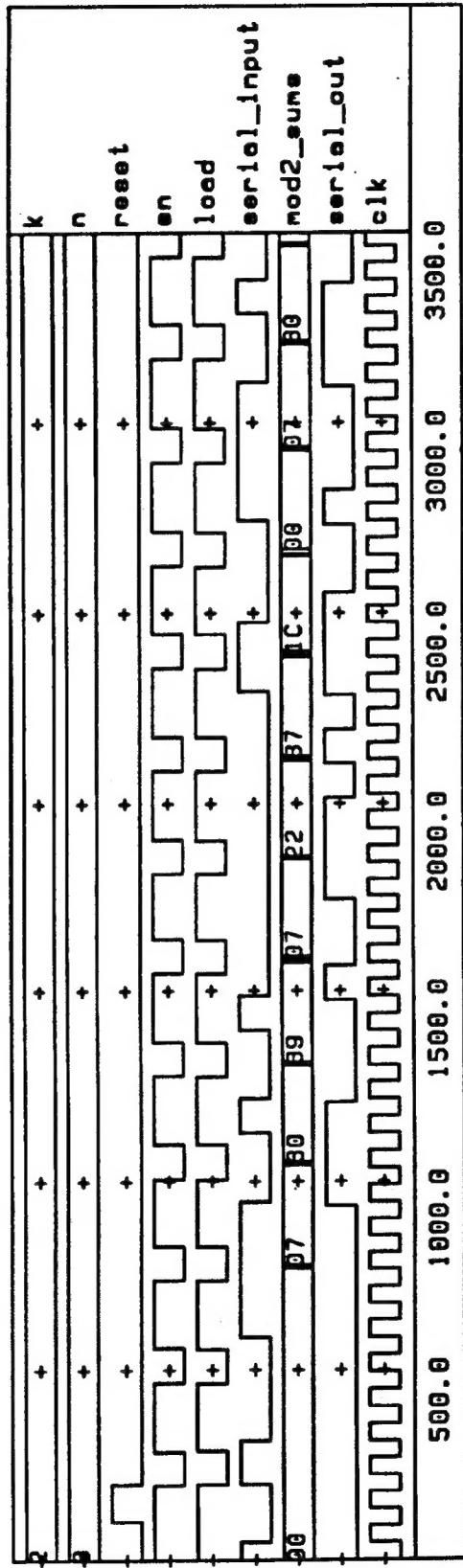
Figure F.1. Output waveform of back annotated design set up for rate 2/3 code.

108

LIST OF REFERENCES

Brown, Stephen D., and others, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, Norwell, MA, 1992.

Clements, Alan, *Microprocessor System Design, 68000 Hardware, software, and Interfacing*, second edition, PWS-Kent Publishing Company, New York, NY, 1992.

Harr, Randolph, et al, *Applications of VHDL to Circuit Design*, Kluwer Academic Publishers, Norwell, MA, 1991.

Klein, B., "Use LFSRs to Build Fast FPGA-Based Counters," *Electronic Design Magazine*, 21 March 1994.

Knapp, Steven K., "Accelerate FPGA Macros with One-Hot Approach," *Electronic Design Magazine*, 13 September 1990.

Lautzenheiser, Dave, and Ravel, Richard B., *Introductory Application Note: Basic Design Flow*, Xilinx, Inc., San Jose, CA, 1989.

Lee, Chin-Hwa, *Digital System Design Using VHDL*, CorralTek, Salinas, CA, 1992.

Lipsett, Roger, et al, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.

Messa, Norman C., *Design Implementation into Field Programmable Gate Arrays*, M.S. Thesis, Naval Postgraduate School, Monterey, California, March 1991.

New, Bernie, "LCA Speed Estimation: Asking the Right Question," *The Programmable Logic Data Book*, Xilinx, Inc., San Jose, CA, 1994.

Proakis, John G., *Digital Communications*, 2nd edition, McGraw-Hill, Inc., New York, NY, 1989.

Simpson, Ken, and Fawcett, Brad, *Design Implementation Application Note: Advanced Design Methodology*, Xilinx, Inc., San Jose, CA, 1989.

Sklar, Bernard, *Digital Communications, Fundamentals and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1988.

Stallings, William, *Data and Computer Communications,* 4th edition, Macmillan Publishing Company, New York, NY, 1994.

Xilinx, Inc., San Jose, CA, *The Programmable Logic Data Book*, 1994.

Xilinx, Inc., San Jose, CA, *2000/3000 Design Implementation Reference Guide*, 1991.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
    Cameron Station
    Alexandria, VA 22304-6145

2. Library, Code 52   2
    Naval Postgraduate School
    Monterey, CA 93943-5101

3. Chairman, Code EC   1
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, CA 93943-5121

4. Chin-Hwa Lee, Code EC/Le   3
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, CA 93943-5121

5. Herschel H. Loomis, Code EC/Lm   3
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, CA 93943-5121

6. Andrew H. Snelgrove   2
    744-D Providence Ave.
    Ventura, CA 93004